D10

```
/**********************************************************************
                                                                 ******
**
** File Name:   RSTinitfin.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**              This module contains the Restore API functions to
**              initialize and terminate the restore operation.
**
** Table of Contents:
** -------------------
**      API Functions:
**              EDMRST_Initialize
**              EDMRST_Finish
**
**      Internal Functions:
**
**
** Compile-Time Options:
**             This section must list any compile time definitions
**             which will affect this header.
**
**********************************************************************
                                                                 *****/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1

/*
 * System headers.
 */
#include <pwd.h>


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>

/*
 * Local headers
 */
#include <RSTinterns.h>
```

```
#include <RSTsup_csm.h>

/*
 * Comms headers.
 */
#include <restore/csc_EDMDispatch.h>
#include <restore/csc_EDMRestoreEng.h>
#include <restore/dispatch_daemon.h>
#include <restore/restore_engine.h>
#include <edmlink/edmlink_api.h>

/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * Global declarations
 */

    internalHandlePtr handlePtr = NULL;
```

```
/**********************************************************************
 * EDMRST_Initialize:
 *
 *  This function takes care of all the initialization for a recovery
 *  session.  This must be called prior to any of the other functions
 *  in the Recover API.
 *
 * Parameters:
 *
 *  hostname   (I) - The machine name of the server to use.
 *  svrHdl       (
             O) - A handle to receive a pointer to this user's client
                    handle for the Restore Engine connection.
 *  timeout    (
         I) - The maximum number of seconds to wait for the connection
 *                  to the Restore Engine process to be completed.
 *
 **********************************************************************/

eerrno_ty
EDMRST_Initialize( hostname_ty   hostname,
                   serverHandle  *svrHdl,
                   unsigned long timeout )
{
        eerrno_ty api_status = E_SUCCESS;

        uid_t   human_uid;
        struct  passwd *pw;
        char    *human_uidname ;

        RE_initialize_args      re_init_args;
        RE_status_result        *re_init_result;
        rpc_if_handle_t                 re_if_spec;
        rpc_binding_handle_t    re_handle;
        int                     retval;
        time_t                  end_time;

#ifdef DEBUG
#define  RPC_TIMEOUT     3600
        struct timeval  rpc_timeout;
#endif

/********************** BEGINNING OF Dispatch Daemon STUFF
                                                   ****************/
    error_status_t status;
    DD_initialize_args    initargs;
    DD_getservicestatus_args  statargs;
    DD_initialize_result  *initres = NULL;
    DD_getservicestatus_result  *statres = NULL;
    rpc_if_handle_t       if_spec;


    time( &end_time );   /* compute time to give up waiting */
    end_time += timeout;

    memset(&if_spec,0,sizeof(rpc_if_handle_t));
    memset(&re_if_spec,0,sizeof(rpc_if_handle_t));

    if (svrHdl == NULL || hostname == NULL )
    {
        return( EP_RB_RECOVER_BAD_ARGS );
    }

        rec_api_log_begin( "edmrestore_api" );
                                        /* init logs, ignore errs?? */
```

```
        /* get user name to pass to DD and RE */
        human_uid = getuid();
        pw = getpwuid( human_uid );
        if (pw == NULL || NULL == pw->pw_name )
        {
                /* Trouble.  */

                rec_api_log_csm(SUB_CSM_USER_NOT_IN_PASSWD, NULL);
                return(EP_RB_RECOVER_PERMISSION_DENIED);
        }

        human_uidname = pw->pw_name;

    handlePtr = (internalHandle *) calloc(1, sizeof(internalHandle));

    /* Use this macro to setup the interface spec */
    CLIENT_IFSPEC(if_spec);

    /* Arrive at a server binding.  Note that if they didn't give us
    ** a valid host parameter, this will fail and drop through and
    ** return NULL in the end.
    ** This call will get and store a fully resolved binding
    ** handle to the host.  The first time we ever call the host,
    ** csc_get_handle will resolve and store the binding.  If we
    ** ever use csc_get_handle to talk to the same host again,
    ** it will just give back the previously resolved binding.
    */

    retval = csc_get_handle((unsigned char *) hostname,
                            if_spec,
                            SERVER_GROUP,
                            &handlePtr -> dd_binding_handle,
                            &status);

    /*
    ** Find out if we got csc handle and see if status is bad.
    ** error_status_ok is a macro defined in cscomm.h.
    */

    if ((status != error_status_ok) || (retval == 0))
    {
      /*  If errno not set, use status if it is a valid errno value */

      if ( errno == 0 )
        errno = ( strerror( status ) ? status : ETIME );

      rec_api_log_csm( SUB_CSM_RPC_FAIL,
                       "failure finding edmdispd to start restore
                                                          engine" );

      return EP_RB_RECOVER_SERVERFAIL;
    }

    errno = 0;

#ifdef DEBUG
/* increase rpc timeout during debugging */
    rpc_timeout.tv_sec = RPC_TIMEOUT;
    rpc_timeout.tv_usec = 0;
    clnt_control( handlePtr->dd_binding_handle, CLSET_TIMEOUT,
                  (char *)&rpc_timeout );
#endif

    initargs.service = DD_SERVICE_RESTORE;
    initargs.hostname = hostname;
    initargs.username = human_uidname;
    initargs.timeout = timeout;

    initres = dd_initialize_1(
                         &initargs, handlePtr -> dd_binding_handle );
            /* Will have _1 for RPC call  */

    if (initres == NULL)
    {
        return EP_RB_RECOVER_RPC_FAIL;
    }

    statargs.service_handle = initres -> service_handle;
    statargs.status = 0;

    statres = dd_getservicestatus_1(
                          &statargs, handlePtr->dd_binding_handle );

    if (statres == NULL)
    {
        return EP_RB_RECOVER_RPC_FAIL;
    }

    while (statres -> status == DD_SERVICE_STARTING )
    {
         time_t  now;

         xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
         time( &now );
         if (now >= end_time)
         {
                 rec_api_log_csm( SUB_CSM_RPC_FAIL,
                        "timeout waiting for edmdispd to start restore
                                                              engine" );
                 return EP_RB_RECOVER_SERVERFAIL;
         }

        sleep(1);

        statres = dd_getservicestatus_1( &statargs,
                                  handlePtr ->
                                          dd_binding_handle );

        if (statres == NULL)
        {
             rec_api_log_csm( SUB_CSM_RPC_FAIL,
       "failure getting status from edmdispd while starting restore
                                                              engine" );
            return EP_RB_RECOVER_RPC_FAIL;
        }
    }

    if (statres -> status != DD_SERVICE_RUNNING)
    {
         rec_api_log_csm( SUB_CSM_RPC_FAIL,
                    "edmdispd failure while starting restore
                                                            engine" );
         xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
        return EP_RB_RECOVER_SERVERFAIL;
    }

    memcpy( handlePtr -> opaque128,
            statres -> handle.handle_val,
            sizeof(handlePtr -> opaque128) );

    xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
```

```
/*********************** END OF Dispatch Daemon STUFF
                                                      ***************/

/* Restore Engine FUNCTIONALITY BEGINS HERE */

/*       RE_CLIENT_IFSPEC(re_if_spec);   */

        retval = csc_private_ifspec_init(
                                   (
                              unsigned char *) handlePtr -> opaque128,
                                RE_PROGNUM,
                                RE_VERSNUM,
                                 &re_if_spec,
                                 &status);

        if (retval == 0)
        {
             rec_api_log_csm( SUB_CSM_RPC_FAIL,
                        "failure initializing interface to restore
                                                            engine" );
            return EP_RB_RECOVER_SERVERFAIL;
        }

         api_status = EP_RB_RECOVER_SERVERFAIL;
         do {
                 time_t  now;
                 time( &now );
                 if (now >= end_time)
                 {
                     rec_api_log_csm( SUB_CSM_RPC_FAIL,
                                   "timeout connecting to restore
                                                            engine" );
                     return EP_RB_RECOVER_SERVERFAIL;
                 }

                 sleep( 1 );
                          /* give restore engine time to get going */
                 retval = csc_connect_to_rpc_service(
                                        (unsigned char *)hostname,
                                        re_if_spec,
                                        RE_CLIENT_GROUP,
                                        &handlePtr ->
                                                   re_binding_handle,
                                        &status);
                 if ((status == error_status_ok) && (retval != 0))
                         api_status = E_SUCCESS;
         } while (api_status != E_SUCCESS);

         if (api_status == E_SUCCESS)
         {
                 re_handle = handlePtr -> re_binding_handle;
#ifdef DEBUG
/*       increase rpc timeout during debugging */
                 rpc_timeout.tv_sec = RPC_TIMEOUT;
                 rpc_timeout.tv_usec = 0;
                 clnt_control( re_handle, CLSET_TIMEOUT, (
                                              char *)&rpc_timeout );
#endif
                 re_init_args.username = human_uidname;
                 set_rpc_obj( re_initialize, &re_init_args.RPCobjID );
                 re_init_result = re_initialize_1(
                                           &re_init_args, re_handle );
                 if (!re_init_result) {
                         api_status = EP_RB_RECOVER_RPC_FAIL;
                         rec_api_log_csm( SUB_CSM_RPC_FAIL,
                                "failure communicating with restore
```

```
                                                                    engine" );
                }
                else {
                        api_status = re_init_result->status;
                        /* release RPC result struct: */
                        xdr_free( xdr_RE_status_result, (
                                                  char *)re_init_result);
                }
        }

        else
            rec_api_log_csm( SUB_CSM_RPC_FAIL,
                                "failure connecting to restore engine" );

        if (
  api_status == E_SUCCESS)    /* return rest eng handle on success */
                *svrHdl = (serverHandle)re_handle;

        return( api_status );

}       /* End of EDMRST_Initialize() */
```

```
/***********************************************************************
 * Ping:
 *
 *  This function allows a ping to be issued in order to keep the
 *  engine alive and running so that the engine will not time out.
 *
 * Parameters:
 *
 *  svrHdl (I) - A pointer to this user's client handle for the
 *               Restore Engine (server) connection.
 *
 ***********************************************************************/
eerrno_ty EDMRST_Ping( serverHandle svrHdl )
{
  eerrno_ty       api_status = E_SUCCESS;
  RE_null_args    re_ping_args;
  RE_status_result        *re_ping_result = NULL;


  if ( NULL == svrHdl || NULL == handlePtr
       || svrHdl != handlePtr->re_binding_handle )
    {
      return( EP_RB_RECOVER_BAD_ARGS );
    }

  set_rpc_obj( re_ping, &re_ping_args.RPCobjID );
  re_ping_result = re_ping_1( &re_ping_args, svrHdl );
  if (NULL == re_ping_result) {
    api_status = EP_RB_RECOVER_RPC_FAIL;
    rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
  }
  else {
    api_status = re_ping_result->status;
    /* release RPC result struct: */
    xdr_free( xdr_RE_status_result, (char *)re_ping_result);
  }

}
```

```
/***********************************************************************
 * EDMRST_Finish
 *
 *    Function Description:
 *
 *  This function terminates a restoral session,
                                              but only during the browse and
 *  mark phase.
        It will be rejected if a restore is currently being executed.
 *  This routine will clean up any local memory used in the session
                                                                   and will
 *  disconnect from the Restore Engine.  After calling this function,
 *  EDMRST_Initialize MUST be called before calling any other
                                                         functions in this
 *  API.
 *
 * Parameters:
 *
 *  svrHdl (I) - A pointer to this user's client handle for the
 *               Restore Engine (server) connection.
 *
 *    Return Codes:
 *       EP_RB_RECOVER_BAD_ARGS
 *       EP_RB_RECOVER_RPC_FAIL
 *       EP_RB_RECOVER_INVALOP
 *       EP_RB_RECOVER_SERVERFAIL
 *
 */

eerrno_ty
EDMRST_Finish( serverHandle svrHdl )
{
        eerrno_ty       api_status = E_SUCCESS;
        RE_null_args    re_finish_args;
        RE_status_result        *re_finish_result = NULL;
        int             csc_status;

        if ( NULL == svrHdl || NULL == handlePtr
          || svrHdl != handlePtr->re_binding_handle )
        {
                return( EP_RB_RECOVER_BAD_ARGS );
        }

        set_rpc_obj( re_finish, &re_finish_args.RPCobjID );
        re_finish_result = re_finish_1( &re_finish_args, svrHdl );
        if (!re_finish_result) {
                api_status = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {
                api_status = re_finish_result->status;
                /* release RPC result struct: */
                xdr_free( xdr_RE_status_result, (
                                        char *)re_finish_result);
        }

        rec_api_log_end();
                        /* write last log and close the log file. */


        return( api_status );

}       /* EDMRST_Finish */
```

```
%/*
%** Copyright 1997,1998 EMC Corporation
%*/

/*
**  Leading % causes rpcgen to pass a line directly thought to the output,
**  ie edmlink_sunrpc.h in this case.  This allows the .h to make a little
**  more sense and be properly documented.
*/

%/*
% * dispatch_daemon.x : EDM Dispatch Daemon C/S communication module
% *
% * Mission Statement:  This is an RPCGEN file which defines the RPC interface
% *                     between the Dispatch Daemon (which resides on
% *                     the EDM server) and  the backup client callers of its
% *                     functions.  This defines the RPC level calls that a
% *                     "caller" can make and the "service" will respond to.
% *
% * Primary Data Acted On: This defines the data that will flow over the wire.
% *                        The RPC mechanism will take care of data
                                                              marshalling
% *
% *
% * Compile-Time Options:
% *                     This acutally gets run through RPCGEN not compiled.
                                                                        It
% *                      must be run through with the -h flag to create a
% *                      header, the -m flag to create the service side
% *                      routines, the -l flag to create the client side
% *                      routines, and the -c flag to create the common data
% *                      marshalling routines.
% *
% * Basic idea here:
% *                     Define the RPC level interfaces to the Dispatch Daemon
% *                     and all data types that will be passed via RPC.
%*/


/*********************************************************************
 Constant Definitions
*********************************************************************/


/*********************************************************************
 Data Structure Definitions
*********************************************************************/

        struct DD_rpc_objID
        {
          int           type;    /* Object identifier (DD_OTYPE_*) */
%#define DD_OTYPE_INIT_IN       1     /* Initialize Input Object */
%#define DD_OTYPE_INIT_OUT      2     /* Initialize Output Object */
          long          len;     /* Length of structure, version number */
        };


        struct DD_client_session_id {
                unsigned long   high;
                unsigned long   low;
        };

const DD_SERVICE_RESTORE=1;
/* structures for input and output of re_initialize rpc call:  */
        struct DD_initialize_args {
                int      service;
```

```
                string hostname<>;
                string username<>;
                unsigned int timeout;
        };

const DD_SERVICE_FAILURE_NONEXEC=-4;
const DD_SERVICE_FAILURE_PERMS=-2;
const DD_SERVICE_FAILURE_EXEC=-1;
const DD_SERVICE_STARTING=1;
const DD_SERVICE_RUNNING=2;
const DD_SERVICE_COMPLETED=4;

        struct DD_initialize_result {
                unsigned int status;
                DD_client_session_id service_handle;
        };

/* structures for getstatus function */
        struct DD_getservicestatus_args {
                int             status;
                DD_client_session_id    service_handle;
        };

        struct DD_getservicestatus_result {
                int             status;
                opaque          handle<128>;
        };

/* work item type */
/*
 * These match the rbconfig.h for the most part. There are
 * some extras for identifying NOS workitems.
 */
const FS_BACKUP_TYPE              =       0;
const SHARED_PART_BACKUP_TYPE     =       1;
const SHARED_M_PART_BACKUP_TYPE =         2;
const OFFLINE_DB_TYPE             =       3;
const ONLINE_KICKDB_TYPE          =       4;
const ONLINE_LISTDB_TYPE          =       5;
const DCONN_KICK_TYPE             =       6;
const DCONN_NET_TYPE              =       7;
const DCONN_WRK_TYPE              =       8;

/* length of various buffers */
const MEDNAME_SIZE =    6;
const TRLNAME_SIZE =    16;
const WINAME_SIZE =     64;
const TEMPLNAME_SIZE =  64;
const USERNAME_SIZE =   64;
const HOSTNAME_SIZE =   256;
const CLNTNAME_SIZE =   64;
const SERVER_SIZE =     256;
const MAX_STRING_SIZE = 256;  /* must be the length of the longest buffer */

/* defines for operation_type */
const BACKUP_TYPE =       1;
const RESTORE_TYPE =      2;
const OTHER_TYPE =        16;

/* work item structure */
struct WIProgress {
    unsigned long       time_started;
    unsigned long       curr_time;

    unsigned long       total_kbytes_sofar;
    unsigned long       total_files;
    unsigned long       total_badfiles;

    unsigned long       curr_kbytes_sofar;
    unsigned long       curr_time_slice;
    unsigned long       curr_files;

    unsigned long       total_files_expected;
    unsigned long       total_kb_expected;

    int                 operation_type;
    int                 completed;
    unsigned long       status;

    struct WIProgress     *next;

    char                wi_name[WINAME_SIZE];
    char                trail_name[TRLNAME_SIZE];
    char                trailset_name[TRLNAME_SIZE];
    char                template_name[TEMPLNAME_SIZE];
    char                client_name[CLNTNAME_SIZE];
    char                server_name[SERVER_SIZE];
    char                media_type[MEDNAME_SIZE];
    char                userid[USERNAME_SIZE];

    char                level;
    char                type;
};

/* SUMMARY structure */
struct EDMProgress {
    unsigned long       time_started;
    unsigned long       curr_time;

    unsigned long       total_kbytes_sofar;
    unsigned long       total_files;
    unsigned long       total_badfiles;

    unsigned long       curr_time_slice;
    unsigned long       curr_kbytes_sofar;
    unsigned long       curr_files;

    unsigned long       active;
    unsigned long       total;
    unsigned long       failed;
    unsigned long       successful;

    unsigned long       total_files_expected;
    unsigned long       total_kb_expected;

    int                 operation_type;
    int                 completed;
    unsigned long       status;

    struct EDMProgress    *next;

    char                host_name[HOSTNAME_SIZE];
};

struct EDMStats
{
    unsigned long        status;
    EDMProgress          edm;
    WIProgress           *wiprogress;
};

struct CC_Notify
```

```
{
    int                 msgtype;
    int                 sourcemodule;
    int                 level;
    int                 msglen;
    string              msgtext<>;
};

struct SessionInfo
{
        DD_client_session_id    service_handle;
        unsigned int            status;
        unsigned long           jobstarttime;
        int                     operation_type;
        long                    lastSent;
        long                    lastReceived;
        int                     outhandle;
        int                     errhandle;

       SessionInfo              *next;
};

struct SessionBlock
{
        struct SessionInfo      *sess;
        int                     totalsessions;
};

program EDM_DISPATCH_DAEMON {

   version EDMDD_FUNCTIONS {

              /* Functions for EDMRST_Initialize */
              DD_initialize_result dd_initialize( DD_initialize_args ) = 1;

              DD_getservicestatus_result dd_getservicestatus(
                                                DD_getservicestatus_args ) = 2;

              SessionBlock dd_getsessioninfo( DD_getservicestatus_args ) = 3;

   } = 1; /* This is version 1 */

%/* This is the RPC program number.
                                   These are reserved in /pds/docs/RPC_numbers
% * This number cannot be re-used by any other RPC daemon on the machine,
                                                                        as it
% * identifies this daemon uniquely.  If it were to be re-used,
                                                               the last daemon
% * to register would be contacted when RPC's come in for this number.
%*/
} = 390015;
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/*
** EDMDispatch.c
**
** Mission Statement: This is the main service file for the dispatch
                                                              daemon.
**                    This file contains the callbacks from the main
                                                            function
**                    which prepares the daemon to go off and service
                                                               RPC's.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      None.
**
** Basic idea here: Module for UNIX specific daemon initialization
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispatch.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <stdarg.h>
#include <string.h>
#include <syslog.h>
#include <pthread.h>
#include <thread.h>
#include <sys/resource.h>

#include <logging/logging.h>
#include <util/esl_core.h>
#include <util/esl_pidfile.h>
#include <util/esl_daemon.h>
#include <csc/cscomm.h>

#include <restore/csc_EDMDispatch.h>

#include <EDMmain.h>
#include <EDMDD_ccw.h>
#include <EDMDispatchLog.h>
#include <EDMDispatchBackground.h>
#include <EDMDDcr_rstsvc.h>

/*
 * Need to define _XOPEN_SOURCE for signal funtion definitions
 * and certain signal structure definitions.
 */
#define _XOPEN_SOURCE
```

```
#include <signal.h>

#undef _XOPEN_SOURCE

static rpc_if_handle_t if_spec;

static int        G_debug = FALSE;
                              /* Variable which will disable forking */

static char      **commandlineargs; /* Pointer to command line args */

/***********************************************************************
                                                             ********
**
** Routine: IsDebugOn
**
** Inputs: None
**
** Outputs:  None
**
** Return Codes:
**              TRUE if debug is on.
**
** Purpose:     This routine can be used to tell other subsystems
**              whether debugging is available.
**
** Intended caller:  internal only.
**
************************************************************************
                                                             ********
*/
boolean_ty
IsDebugOn()
{
    return G_debug;
}
```

```c
/*********************************************************************
                                                              ********
**
** Routine: kill_handler
**
** Inputs: int sigval - the signal which was receieved.
**
** Outputs:  Will log messages telling what action is being taken.
**
** Return Codes:
**                exits with the number of the signal received
**
** Purpose:       This routine handles specific signals i.e. SIGINT,
**                SIGQUIT,
                       SIGTERM. Each results in a log entry and an exit.
**
** Intended caller:  internal only.
**
*********************************************************************
                                                              ********
*/
static void kill_handler( IN int sigval )
{
int       status;
time_t    current_time;
char      *ctimebuf;
char      *ebuff;

  /* If main exits, it calls this routine with signal 0 */

  /* Unregister the interface */
  (void) csc_unregister_server_interface(&if_spec, &status);

  /* If the unregister fails, report the problem, but continue */
  if ( status != error_status_ok )
  {
    ebuff = (char *) csc_get_error( status );

    (void) EDMDispatch_logent(
                    __FILE__,__LINE__, LOG_ERR, MESSAGE_NO_LOGIN, 0,
                       "CSC_SERVER_LOGIN failed: <%d> %s",
                       status, (ebuff ? ebuff : "Unknown error") );
  }

  /* Get the current time */
  (void) time(&current_time);

  ctimebuf = ctime(&current_time);

  /* Overlay newline with null - buf should always be 26 bytes long */
  ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

  (void) EDMDispatch_logent(
                    __FILE__,__LINE__, LOG_INFO, MESSAGE_SHUTDOWN, 0,
                    "Shutting down at %s due to signal %d", ctimebuf,
                                                          sigval);

  /*
   * Remove our lock file.
   */
  (void) EslDestroyPidFile(PIDPATH);

  exit(sigval);

} /* End of kill_handler() */
```

```c
/*********************************************************************
                                                              ********
 * Function Name:
 *       display_usage
 *
 * Simply displays the usage
 *
 * Call Arguments:
 *       Program name
 *
 * Error Outputs and Side Effects:
 *       Prints usage.
 *
 * Special Considerations:
 *       None.
 *
 *********************************************************************
                                                              ********
 */
static void
display_usage (IN char *progname)
{
  /* Print out usage stmt.  */

  fprintf (stderr,"Usage: %s [-d]\n", progname);
  fprintf (
   stderr,"-d keep the daemon from forking so debugging is easier\n");

} /* end display_usage () */
```

```
/***********************************************************************
                                                                ********
**
** Routine: daemon_catch_interrupts
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              None
**
** Purpose:     Sets up signals for service. On NT we will have to
**              consider what OS constructs to replace signals with.
**              In this case we are catching SIGTERM, SIGINT, and
**              SIGQUIT and ignoring anything else.
**
** Intended caller:  internal only.
**
************************************************************************
                                                                ********
*/
void daemon_catch_interrupts()
{
struct sigaction         sactions;       /* Signal actions */

          ZERO( sactions );

         /*
          * Set an empty list so we can set signals we want to handle
          */
          (void) sigemptyset( &sactions.sa_mask );

         /*
          * Add signals that we want to handle
          */
          (void) sigaddset( &sactions.sa_mask, SIGTERM );
          (void) sigaddset( &sactions.sa_mask, SIGINT );
          (void) sigaddset( &sactions.sa_mask, SIGQUIT );

          /* Setup the signal handler. */
          sactions.sa_handler = kill_handler;

         /*
          * Assign handler to each signal we are interested in.
          */
          (void) sigaction( SIGTERM, &sactions, NULL );
          (void) sigaction( SIGINT,  &sactions, NULL );
          (void) sigaction( SIGQUIT, &sactions, NULL );

          /*
          * Setup mask so we can specify what signals we will ignore.
          */
          (void) sigfillset( &sactions.sa_mask );

         /*
          * We want to ignore everything except those we have set up
          * above so remove those from the list.
          */
          (void) sigdelset( &sactions.sa_mask, SIGTERM );
          (void) sigdelset( &sactions.sa_mask, SIGINT );
          (void) sigdelset( &sactions.sa_mask, SIGQUIT );

         /*
```

```
          * Set the mask. Since no other threads have been started,
          * all threads will get this mask.
          */
          (void) thr_sigsetmask( SIG_SETMASK, &sactions.sa_mask, NULL );
}
```

```
/**********************************************************************
                                                              ********
**
** Routine: daemon_check_proper_ID
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              exits with an error when the user is not root
**
** Purpose:     Checks user's ID and determines if the user is allowed
**              to execute service.
                                If there are no constraints then this
**              function may be blank.
**
** Intended caller:  internal only.
**
**********************************************************************
                                                              ********
*/

void daemon_check_proper_ID()
{
        /*
        ** Check for root
        */

        if (geteuid() != E_ROOTUID)
        {
          (void) EDMDispatch_logent(
                  __FILE__,__LINE__, LOG_ERR, DAEMON_NOTSUPERUSER, 0,
                          "Must be run as superuser, uid was %d",
                          geteuid());
          exit(1);
        }
}
```

```
/**********************************************************************
                                                              ********
**
** Routine: parse_commandline
**
** Inputs:      argc, argv (command line arguments)
**
** Outputs:     None
**
** Return Codes:
**              exits with an error when the user types a bad argument
**
** Purpose:     Parses command line arguments and sets flags. If there
**              are no flags to be set then this function may be empty.
**
** Intended caller:  internal only.
**
**********************************************************************
                                                              ********
*/

void parse_commandline(int argc, char *argv[])
{
int                     opt;            /* Process options    */

        commandlineargs = argv;

        while ((opt = getopt(argc,argv,"dD")) != EOF )
        {
          switch(opt)
          {
            case 'd':
            case 'D':
                G_debug = TRUE;
                break;
            default:
                (void) display_usage( argv[0] );
                exit(1);

          }
        }
}
```

```
/**********************************************************************
                                                                ********
**
** Routine: daemon_initialize_logging
**
** Inputs:       None
**
** Outputs:      None
**
** Return Codes:
**               None
**
** Purpose:      Do whatever it takes to initialize logging.
                                                      In the near
**               future this may involve doing something with catalogs
                                                               or
**               calling higher level logging functions which
                                                      encapsulate
**               these things.
**
** Intended caller:  internal only.
**
**********************************************************************
                                                                ********
*/

void
daemon_initialize_logging()
{
        /* Pass in argv[0], the program name */
        (void) esl_log_init(commandlineargs[0]);
}
```

```
/**********************************************************************
                                                                ********
**
** Routine: daemon_become_daemon
**
** Inputs:       None
**
** Outputs:      None
**
** Return Codes:
**               exits with an error code if initialization fails
**
** Purpose:      This function is for doing the forking etc. under UNIX.
**               It is unknown what will be necessary under NT.
**
** Intended caller:  internal only.
**
**********************************************************************
                                                                ********
*/

void
daemon_become_daemon()
{
        char *ptr;
        int  ret = 0;

        /*
         * Strip the path from the program name so we can use it
         * elsewhere.
         */
        ptr = strrchr(commandlineargs[0], '/');
        if (ptr == NULL)
            ptr = commandlineargs[0];
        else
            ptr++;

        /* Change directory to a process specific core directory */
        ret = esl_coredir_setup(ptr);
        if (ret != 0)
        {
             (void) EDMDispatch_logent( __FILE__,__LINE__,LOG_ERR,
                             MESSAGE_ERR_IN_ESL_COREDIR, 0,
                             "esl_coredir_setup failed: <%d>",
                             errno);
            exit(1);
        }

         /*
         **  This is now esl functionality.
                                   This code does everything necessary
         **  to make this a "real" daemon by detaching from the
                                                              terminal
         **  changing the process group, closing stdout, stderr, stdin,
                                                                   ...
         */

        if (G_debug == FALSE)
        {
            ret = esl_daemon_startup();
             if (ret != 0)
            {
                fprintf(
                      stderr, "%s: Failed to initialize as daemon.\n",
                        commandlineargs[0]);
                        exit(1);
                }
        }
}
```

```
/*************************************************************************
                                                                 ********
**
** Routine: rpc_init
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              exits with an error code if initialization fails
**
** Purpose:     This function is for doing RPC initialization.
**              For the most part it involves calling the csc routines.
**              This is pretty standard between UNIX and NT.
**
** Intended caller:  internal only.
**
**************************************************************************
                                                                 ********
*/

void rpc_init()
{
error_status_t          status;
                                                /* error status (nbase.h) */
char   *ebuff;

/*
** This is here because of HP which may or may not define timeval.
** May be removed when esl_timeval is ported to clients
*/
#ifdef _STRUCT_TIMEVAL
        struct timeval  sleep_interval = {5,0};
                                                /* 5 second sleep interval */
#else
        struct timespec sleep_interval = {5,0};
                                                /* 5 second sleep interval */
#endif

        /* Setup the interface specification for RPC */
        SERVER_IFSPEC(if_spec);

        /*
        * Login as SERVER_PRINCIPAL.  The context of the process
        * will be set to this principal.
        *
        * This process will keep trying to login to DCE if the
                                                                registery
        * server is unavailable.
                                Note that under SUN RPC this is a no-op.
        */
        while (TRUE)
        {
                (void) csc_server_login(SERVER_PRINCIPAL,
                                        SERVER_KEYTAB, &status);

                /* If we succeeded, then exit this loop.  */
                if ( status == error_status_ok )
                {
                        break;
                }
                else    /* Print error message if appropriate. */
                {
                 ebuff = (char *) csc_get_error( status );
                (void) EDMDispatch_logent(
                                        __FILE__,__LINE__,LOG_ERR,
                                 MESSAGE_NO_LOGIN, 0,
                                 "CSC_SERVER_LOGIN failed: <%d>
                                                           %s",
                                 status, (
                               ebuff ? ebuff : "Unknown error"));
            }

            /* If the failure was due to unavailable client,
            * pause and then try again.
            */
            if (status == sec_rgy_server_unavailable)
            {
                    /*
                     * uses sleep when SUNRPC, otherwise uses
                     * pthread call to delay for the specified
                     * time
                     */
                      CSC_SLEEP(sleep_interval);
                      continue;
            }

            /* If we got here, we had a unexpected failure.  */
            (void) EDMDispatch_logent( __FILE__,__LINE__,LOG_ERR,
                               MESSAGE_NO_LOGIN, 0,
                              "The service cannot log in as
                                                   required");

            exit(1);
    }

    /*
    ** We need to initialize the authorization module before we
                                                              do
    ** a listen.
    */
    (void)csc_authorization_init(&status);

    if ( status != error_status_ok )
    {
     ebuff = (char *) csc_get_error( status );

      (void) EDMDispatch_logent( __FILE__,__LINE__,LOG_ERR,
                         MESSAGE_NOAUTHORIZATION, 0,
                         "CSC_AUTHORIZATION_INIT failed: <%d> %s",
                          status, (
                               ebuff ? ebuff : "Unknown error") );
      exit(1);
    }


    (void) csc_register_server_interface(   &if_spec,
                                            SERVER_ANNOTATION,
                                            &status);

    if ( status != error_status_ok )
    {
     ebuff = (char *) csc_get_error( status );

      (void) EDMDispatch_logent( __FILE__,__LINE__,LOG_ERR,
                          MESSAGE_CANNOTREGISTER, 0,
                         "CSC_REGISTER_SERVER_INTERFACE failed:
                                                     <%d> %s",
                          status, (
```

```
                               ebuff ? ebuff : "Unknown error") );
              exit(1);
           }
}
```

```
/*********************************************************************
                                                             ********
**
** Routine: rpc_run
**
** Inputs:       None
**
** Outputs:      None
**
** Return Codes:
**               None
**
** Purpose:      This function is for running the RPC listen.
**               This is pretty standard between UNIX and NT.
**
** Intended caller:  internal only.
**
*********************************************************************
                                                             ********
*/

void rpc_run()
{
error_status_t            status;
                                        /* error status (nbase.h) */
char  *ebuff;

        /* listen for RPC calls forever. */
        (void) csc_server_listen(
                        rpc_c_listen_max_calls_default, &status );

       ebuff = (char *) csc_get_error( status );

        /* We don't expect to get here. */
        (void) EDMDispatch_logent(
                   __FILE__,__LINE__,LOG_ERR, MESSAGE_SERVERLISTEN, 0,
                         "CSC_SERVER_LISTEN failed: <%d> %s",
                          status, (
                                ebuff ? ebuff : "Unknown error") );
}
```

```
/*********************************************************************
                                                             ********
**
** Routine: daemon_specific_initialization
**
** Inputs:       None
**
** Outputs:      None
**
** Return Codes:
**               None
**
** Purpose:      Do whatever makes this daemon special.
                                                      In some cases you
**               may want to start a thread or open a socket.
                                                           Do that here.
**
** Intended caller:  internal only.
**
*********************************************************************
                                                             ********
*/

void
daemon_specific_initialization()
{
error_status_t            status;
                                        /* error status (nbase.h) */
int                       ret;
pthread_t                 mantid;
pthread_t                 cleantid;
time_t   current_time;
char     *ctimebuf;
struct rlimit rlp;

        /* Create a file and lock it so we don't start multiple
         ** daemons.  Exit if there is another copy of us running.
         ** The CreatePidFile call already logs errors so just exit.
         */
        if (EslCreatePidFile(PIDPATH))
        {
            exit(1);
        }

         /* Find out what time it is */
         (void) time(&current_time);

         ctimebuf = ctime(&current_time);

         /* Overlay newline with null - buf should always be 26 bytes
                                                                  long */

         ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

         /* Log startup message */
         (void) EDMDispatch_logent(
                    __FILE__,__LINE__, LOG_INFO, MESSAGE_STARTUP, 0,
                         "Restore service %s starting up at %s",
                                                  commandlineargs[0],
                         ctimebuf );

        /* set the open files limit very large */
        rlp.rlim_max = FD_SETSIZE;
        rlp.rlim_cur = FD_SETSIZE;

        setrlimit(RLIMIT_NOFILE, &rlp);
        getrlimit(RLIMIT_NOFILE, &rlp);

        (void) EDMDispatch_logent(
                    __FILE__,__LINE__, LOG_INFO, MESSAGE_STARTUP, 0,
                        "Service allows %d open files",
                                                rlp.rlim_max );

        /* Initialize service launcher */
        ret = EDMDDSvcInit();

        if (ret != 0)
        {
            (void) EDMDispatch_logent(
                                __FILE__,__LINE__, LOG_INFO, 0, 0,
                        "Service Launcher failed returning - %d",
                                                        ret);
            exit(1);
        }

        /*
         * Start the other threads in the daemon. The main thread
         * becomes the RPC thread. BAMDataManage is the entry point
         * for the data collection thread. BAMDataCleanup is the
         * entry point for the data expiration thread.
         */
        /*pthread_create(&mantid, NULL, DispDaemon_ccr, NULL); */
        pthread_create(&mantid, NULL, DispDaemon_ccw, NULL);
        pthread_create(&cleantid, NULL, DispatchBackground, NULL);
        rpc_init();
        rpc_run();
}
```

```
/***********************************************************************
                                                                ********
**
** Routine: daemon_cleanup
**
** Inputs:      None
**
** Outputs:     None
**
** Return Codes:
**              None
**
** Purpose:     Call function which will clean up daemon properly.
**
** Intended caller:  internal only.
**
************************************************************************
                                                                ********
*/

void
daemon_cleanup()
{
            kill_handler( 0 );
}
```

```
/*
** Copyright 1996,1997 EMC Corporation
*/

/* EDMDispatchService.c
 *
 *
 * Mission Statement:  RPC entry points.
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
 */

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchService.c,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/inout.h>

#include <logging/logging.h>
#include <csc/cscomm.h>

#include <restore/csc_EDMDispatch.h>
#include <restore/dispatch_daemon.h>

#include <EDMDispatchLog.h>
#include <EDMDispatchSession.h>

/*
 * These are all the rpc entry points for the dispatch daemon.
 * The dispatch daemon is multi-threaded and it is the main thread
 * which handles all incoming RPC. ONC RPC is single threaded
 * so each call blocks other RPC calls. This provides us some
 * safety in the way we handle our data and limits our exposure
 * to unexpected multithreading problems.
 */
static void FreeSessionInfo(SessionInfo *);

/*********************************************************************
                                                                ********
**
** Routine:  dd_initialize_1
**
** Inputs:   DD_initialize_args * - args for the restore initialize
                                                                   call
**
** Outputs:  None
**
** Return Codes:
**           DD_initialize_result * - result of init function call
**
** Purpose:  Function to create a restore session.
**
** Intended caller:  Internal Only.
*********************************************************************
                                                                ********
*/

DD_initialize_result *
```

```
dd_initialize_1_svc(
                   IN DD_initialize_args *arg, IN struct svc_req *req )
{
    static DD_initialize_result argzz;

    InitializeSession(arg, req, &argzz);

    return &argzz;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  dd_getservicestatus_1
**
** Inputs:   DD_getservicestatus_args * - args for the
                                          getservicestatus call
**
** Outputs:  None
**
** Return Codes:
**           DD_getservicestatus_result * - result of status function
                                                                 call
**
** Purpose:  Function to poll for status on a session.
**
** Intended caller:  Internal Only.
*********************************************************************
                                                              ********
*/

DD_getservicestatus_result *
dd_getservicestatus_1_svc(
          IN DD_getservicestatus_args *arg, IN struct svc_req *req )
{
    static DD_getservicestatus_result argzz;

    GetDispatchStatus(arg, &argzz);

    return &argzz;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  dd_getsessioninfo_1
**
** Inputs:   DD_getservicestatus_args * - args for the getsessioninfo
                                                                 call
**
** Outputs:  None
**
** Return Codes:
**           SessionBlock * - result of session info call
**
** Purpose:  Function to get information on all sessions.
**
** Intended caller:  Internal Only.
*********************************************************************
                                                              ********
*/

SessionBlock *
dd_getsessioninfo_1_svc(
          IN DD_getservicestatus_args *arg, IN struct svc_req *req )
{
    static SessionBlock argzz;
    static boolean_ty first = TRUE;

    if (first)
    {
        memset(&argzz, 0, sizeof(argzz));
        first = FALSE;
    }
    else
    {
        FreeSessionInfo(argzz.sess);
        argzz.sess = NULL;
    }

    GetDispatchInfo(arg, &argzz);

    return &argzz;
}
```

```
/*******************************************************************
                                                           ********
**
** Routine:  FreeSessionInfo
**
** Inputs:   SessionInfo * - arg to free
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Function to free all SessionInfo structures in a list.
**
** Intended caller:  Internal Only.
********************************************************************
                                                          *********/

static void FreeSessionInfo(SessionInfo *sess)
{
    if (sess == NULL)
        return;

    if (sess -> next != NULL)
        FreeSessionInfo(sess -> next);

    free(sess);
}
```

```
/*
** Copyright 1996,1999 EMC Corporation
*/

/*
** EDMDispatchSession.cc
**
** Mission Statement: This is where all session management occurs.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc
                                                         support.  If
**                                     not set, assume DCE support.
**
** Basic idea here: Module for session management
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchSession.cc,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <pthread.h>
#include <memory.h>
#include <sys/time.h>
#include <sys/types.h>
#include <syslog.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>

#include <csc/cscomm.h>
#include <restore/dispatch_daemon.h>
#include <restore/dispatch_protocol_client.h>
#include <EDMSession.h>
#include <EDMReturnMessageApi.h>
#include <EDMDHandleMgrApi.h>
#include <EDMDispatchSession.h>
#include <EDMDispatchConfig.h>
#include <EDMDDcr_rstsvc.h>

#include <EDMDispatchLog.h>

static RWBinaryTree      G_sessionTree;

static pthread_mutex_t    G_sessionTreeMtx = PTHREAD_MUTEX_INITIALIZER;
extern ELinkHandlePtr_ty  ELinkHandle;
```

```
static int maxDisconnectTime = SECONDS_PER_HOUR; // one hour

/*********************************************************************
                                                            ********
**
** Routine:  LockSessionMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the session mutex.
**
*********************************************************************
                                                            ********
*/

static void
LockSessionMutex()
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_sessionTreeMtx, NULL);
    }

    pthread_mutex_lock(&G_sessionTreeMtx);
}
```

```
/*******************************************************************
                                                            ********
**
** Routine:  UnlockSessionMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Unlock the mutex for the session tree object
**
*******************************************************************
                                                            ********
*/

static void
UnlockSessionMutex()
{
    pthread_mutex_unlock(&G_sessionTreeMtx);
}
```

```
/*******************************************************************
                                                            ********
**
** Routine:  InitializeSession
**
** Inputs:   DD_initialize_args *arg - args sent via RPC for starting
                                                             session
**           struct svc_req *req - the request block from RPC
**
** Outputs:  DD_initialize_result *res - the result structure which
                                                        tells whether
**                     operation succeeded or failed.
**
** Return Codes:
**           None
**
** Purpose:  Initialize a session for the GUI.
**
*******************************************************************
                                                            ********
*/

void
InitializeSession(IN DD_initialize_args *arg, IN struct svc_req *req,
                  OUT DD_initialize_result *res)
{
    EDMSession  *session;
    EDMSession  *ret;
    pthread_t   id;
    time_t       t;

    if (arg == NULL || req == NULL || res == NULL)
    {
        return;
    }

    t = time(NULL);

    session = new EDMSession();

    if (session == NULL)
    {
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        return;
    }

    session -> initSession();

    session -> setStartTime(t);

    session -> setOperationType(arg -> service);

    session -> setStatus(DD_SERVICE_STARTING);

    if (arg -> username  != NULL && arg -> hostname != NULL)
    {
      switch(arg -> service)
        {
           // code is commented out because we do not
           // want to read tthe config for permission information
           // at this time, it is a waste of cycles
#if 0
             case DD_SERVICE_RESTORE : boolean_ty allowed;

                                allowed =
                                        DispatchCheckRestorePermission(
                                            arg->hostname,
                                                    arg -> username);

                                    if (!allowed)
                                    {
                                        res -> status =
                                              DD_SERVICE_FAILURE_PERMS;
                                       delete session;
                                        return;
                                    }

                                    break;
#endif
        default: // Add some error messsage for unknown service
                                    break;
        };
    }
    else
    {
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        delete session;
        return;
    }

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.insert((
                                            RWCollectable *) session);

    UnlockSessionMutex();

    if (ret == NULL)
    {
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        delete session;
        return;
    }

    session -> getSessionID(&res -> service_handle);

    // Call Steve's thread
    pthread_create(&id, NULL, &DDRSTsvc_init, (void *) session);

    session -> setThreadID(id);

    return;
}
```

```
/**********************************************************************
                                                              ********
**
** Routine:  SendPingMessagesToSession
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Queue up all the ping messages to the sessions.
                                                      If they don't
**           respond they should be considered dead.
**
***********************************************************************
                                                              ********
*/

void
SendPingMessagesToSession()
{
    EDMSession *sess;

    LockSessionMutex();

    RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                        G_sessionTree);

    while ( sessionIterator != NULL &&
           (sess = (EDMSession*) (*sessionIterator)()) != NULL )
    {
        DD_client_session_id sid;
        rpc_binding_handle_t *cscb = NULL;
        int                    status;
        int                    ret;

        if (sess -> getStatus() != DD_SERVICE_RUNNING)
            continue;

        sess -> getSessionID(&sid);

        ret = GetCSCHandle(&sid, &cscb, &status);

        if (ret != 0 || cscb == NULL || *cscb == NULL)
            continue;

        PushResponseMessage(dp_ping_request, sid, cscb, &status);
    }

    // through with iterator
    if (sessionIterator != NULL)
    {
        delete sessionIterator;
    }

    UnlockSessionMutex();
}
```

```
/*****************************************************************
                                                          ********
**
** Routine:  UpdateSessionLastReceived
**
** Inputs:   DD_client_session_id *sessID - session that sent us
                                                         something
**
** Outputs:  None
**
** Return Codes:
**             0 on success and non-zero otherwise
**
** Purpose:  Update the specified session with the lastest received
                                                           message
**           time.
**
*****************************************************************
                                                          ********
*/

int
UpdateSessionLastReceived(DD_client_session_id *sessID)
{
    time_t   last = time(NULL);
    EDMSession  *session;
    EDMSession  *ret;

    session = new EDMSession();

    if (session == NULL)
    {
        EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                            "Failure to create a session block");
        return -1;
    }

    session -> setSessionID(sessID);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

    UnlockSessionMutex();

    delete session;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                            "Failure to update session %ld:%ld received
                                                             time",
                            sessID -> high, sessID -> low);
        return -1;
    }

    ret -> setLastReceived(last);

    return 0;
}
```

```
/*****************************************************************
                                                          ********
**
** Routine:  UpdateSessionLastSent
**
** Inputs:   DD_client_session_id *sessID - session that sent us
                                                         something
**
** Outputs:  None
**
** Return Codes:
**             0 on success and non-zero otherwise
**
** Purpose:  Update the specified session with the lastest sent
                                                           message
**           time.
**
*****************************************************************
                                                          ********
*/

int
UpdateSessionLastSent(DD_client_session_id *sessID)
{
    time_t   last = time(NULL);
    EDMSession  *session;
    EDMSession  *ret;

    session = new EDMSession();

    if (session == NULL)
    {
        EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                            "Failure to create a session block");
        return -1;
    }

    session -> setSessionID(sessID);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

    UnlockSessionMutex();

    delete session;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                            "Failure to update session %ld:%ld sent
                                                             time",
                            sessID -> high, sessID -> low);
        return -1;
    }

    ret -> setLastSent(last);

    return 0;
}
```

```
/*********************************************************************
                                                              ********
**
** Routine:  CheckDispatchSessions
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Look for dead sessions and kill them off
**
*********************************************************************
                                                              ********
*/

void
CheckDispatchSessions()
{
    EDMSession  *sess;
    int         status = 0;
    int         ret = 0;
    time_t      currTime;
    RWBinaryTree reaperTree;

    currTime = time(NULL);

    LockSessionMutex();

    RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                  G_sessionTree);

    while ( sessionIterator != NULL &&
          (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {

        if ( (sess->getLastReceived(
 ) <= currTime - maxDisconnectTime && sess->getLastReceived() != 0) ||
             (sess->getStartTime() <= currTime - maxDisconnectTime &&
             (sess -> getStatus(
                 ) == DD_SERVICE_FAILURE_NONEXEC || sess -> getStatus(
                                  ) == DD_SERVICE_FAILURE_EXEC ||
              sess -> getStatus() == DD_SERVICE_FAILURE_PERMS)) )
        {
            // Insert it into the reaper tree
            (void) reaperTree.insert(sess);
        }
    }

    // through with iterator
    if (sessionIterator != NULL)
    {
        delete sessionIterator;
    }

    UnlockSessionMutex();

    // If the reaper tree has something in it then use those entries
                                                         to remove
    // things from the query tree.
    if (reaperTree.entries() > 0)
    {
        sessionIterator = new RWBinaryTreeIterator(reaperTree);
```

```
        while ( sessionIterator != NULL &&
              (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {
            DD_client_session_id  sessID;

            sess -> getSessionID(&sessID);

            ret = removeSession(&sessID, &status);

            if (ret != 0)
            {
                EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
                                  "Failure to remove session %ld:%ld",
                                  sessID.high, sessID.low);
                continue;
            }
            else
            {
                EDMDispatch_logent(
                                  __FILE__, __LINE__, LOG_INFO, 0, 0,
                         "Removing session %ld:%ld,
                   Haven't recieved anything since %ld. Current %ld",
                         sessID.high, sessID.low,
                                     sess -> getLastReceived(),
                         currTime - maxDisconnectTime);
            }

            ret = deleteHandleSet(&sessID, &ELinkHandle, &status);

            if (ret != 0)
            {
                EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
                                  "Failure to delete handles for
                                             session %ld:%ld",
                                  sessID.high, sessID.low);
            }
        }

        // through with iterator
        if (sessionIterator != NULL)
        {
            delete sessionIterator;
        }

        reaperTree.clear();
    }
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  DrainSessionDescriptors
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Drain whatever data is on stdout and stderr for sessions.
**
************************************************************************
                                                                ********
*/

void
DrainSessionDescriptors()
{
    int     hout = 0, herr = 0, status = 0;
    int     selret = 0;
    int     i = 0;
    char     buff[1024];
    struct timeval timetowait = {
           1, 0
          };
    fd_set  stdoutSet;
    fd_set  stderrSet;

    getStdoutSet(&stdoutSet, &hout, &status);

    if ( (selret = select(
                   hout + 1, &stdoutSet, NULL, NULL, &timetowait)) >= 0)
    {
        for (; i < hout+1; i++)
        {
            if (FD_ISSET(i, &stdoutSet))
            {
                while (read(i, buff, 1024) > 0);
            }
        }
    }

    getStderrSet(&stderrSet, &herr, &status);

    if ( (selret = select(
                   herr + 1, &stderrSet, NULL, NULL, &timetowait)) >= 0)
    {
        for (i = 0; i < herr+1; i++)
        {
            if (FD_ISSET(i, &stderrSet))
            {
                while (read(i, buff, 1024) > 0);
            }
        }
    }
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  GetSessionStatus
**
** Inputs:   DD_client_session_id *ssid - session ID to check the
                                                             status of
**
** Outputs:  int *status - status of the function call
**           int *s_status - session status
**
** Return Codes:
**           0 if successful and non-zero otherwise
**
** Purpose:  Get status on the session.
**
************************************************************************
                                                                ********
*/
int
GetSessionStatus(
                DD_client_session_id *ssid, int *s_status, int *status)
{
    EDMSession  *sess;
    EDMSession  *ret;

    if (status == NULL)
    {
        return -1;
    }

    if (ssid == NULL || s_status == NULL)
    {
        *status = SESSION_BAD_ARGS;

        return -1;
    }

    sess = new EDMSession();

    if (sess == NULL)
    {
        EDMDispatch_logent(
                 __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                         "Failure to create a session block");
        *status = SESSION_NO_MEMORY;

        return -1;
    }

    sess -> setSessionID(ssid);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

    UnlockSessionMutex();

    delete sess;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                         "Failure to lookup session %ld:%ld",
                                  ssid -> high, ssid -> low);
        *status = SESSION_LOOKUP_FAILED;
        return -1;
    }

    *s_status = ret -> getStatus();

    return 0;
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  GetDispatchStatus
**
** Inputs:   DD_getservicestatus_args *arg - session ID to check the
                                                              status of
**
** Outputs:  DD_getservicestatus_result *res - the result structure
                                                             which tells
**                     whether operation succeeded or failed.
**
** Return Codes:
**           None
**
** Purpose:  Get status on the starting session.
**
************************************************************************
                                                                ********
*/

void
GetDispatchStatus(IN DD_getservicestatus_args *arg,
                  OUT DD_getservicestatus_result *res)
{
    EDMSession  *sess;
    EDMSession  *ret;
    static char buff[CONNECT_HANDLE_SIZE];

    sess = new EDMSession();

    if (sess == NULL)
    { // Give an error
        EDMDispatch_logent(
                  __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                         "Failure to create a session block");
        return;
    }

    sess -> setSessionID(&arg -> service_handle);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

    UnlockSessionMutex();

    delete sess;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                         "Failure to lookup session %ld:%ld",
                         arg -> service_handle.high,
                                 arg -> service_handle.low);

        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        return;
    }

    res -> status = ret -> getStatus();

    memset(buff, 0, sizeof(buff));
    if (res -> status == DD_SERVICE_RUNNING)
    {
        res -> handle.handle_val = (char *) ret -> getConnectionHandle(
                                                                    );
        res -> handle.handle_len = CONNECT_HANDLE_SIZE;
    }
    else
    {
        res -> handle.handle_val = (char *) buff;
        res -> handle.handle_len = CONNECT_HANDLE_SIZE;
    }
}
```

```
/***********************************************************************
                                                                ********
**
** Routine:  GetDispatchInfo
**
** Inputs:   DD_getservicestatus_args *arg - session ID to check the
                                                             status of
**
** Outputs:  SessionBlock *res - the information regarding the
                                                      specified session
**
** Return Codes:
**           None
**
** Purpose:  Get status on all the sessions.
**
************************************************************************
                                                                ********
*/

void
GetDispatchInfo(IN DD_getservicestatus_args *arg,
                OUT SessionBlock *res)
{
    EDMSession   *sess;
    EDMSession   *ret;
    SessionInfo  *sinfo, *slast;
    static char buff[CONNECT_HANDLE_SIZE];

    LockSessionMutex();

    if (arg -> service_handle.high != 0 && arg -> service_handle.
                                                          low != 0)
    {
        // Looking for a single session. Do a find.
        sess = new EDMSession();

        if (sess == NULL)
        { // Give an error
            EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                              "Failure to create a session block");

            UnlockSessionMutex();
            return;
        }

        sess -> setSessionID(&arg -> service_handle);

        ret = (EDMSession *) G_sessionTree.find(sess);

        delete sess;

        if (ret == NULL)
        {
            EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                              "Failure to lookup session %ld:%ld",
                          arg -> service_handle.high,
                                         arg -> service_handle.low);
            UnlockSessionMutex();
            return;
        }

        res -> totalsessions = 1;
        res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

        if (res -> sess == NULL)
        {
            EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                               "Failure to allocate session info
                                                              block");
            UnlockSessionMutex();
            return;
        }

        sinfo = res -> sess;

        ret -> getSessionID(&sinfo -> service_handle);
        sinfo -> status = ret -> getStatus();
        sinfo -> jobstarttime = ret -> getStartTime();
        sinfo -> operation_type = ret -> getOperationType();
        sinfo -> lastSent = ret -> getLastSent();
        sinfo -> lastReceived = ret -> getLastReceived();
    }
    else
    {
        res -> totalsessions = 0;

        res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

        if (res -> sess == NULL)
        {
            EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                               "Failure to allocate session info
                                                              block");
            UnlockSessionMutex();
            return;
        }

        sinfo = res -> sess;

        RWBinaryTreeIterator *sessionIterator = new
                                RWBinaryTreeIterator(G_sessionTree);

        boolean_ty addnext = FALSE;

        while ( sessionIterator != NULL && (ret = (EDMSession*) (
                                  *sessionIterator)()) != NULL )
        {
            int                   status;

            if (addnext)
            {
                sinfo -> next = (SessionInfo *) calloc(1, sizeof(
                                                         SessionInfo));

                if (sinfo -> next == NULL)
                {
                    break;
                }

                sinfo = sinfo -> next;
            }

            ret -> getSessionID(&sinfo -> service_handle);
            sinfo -> status = ret -> getStatus();
            sinfo -> jobstarttime = ret -> getStartTime();
```

```
            sinfo -> operation_type = ret -> getOperationType();
            sinfo -> lastSent = ret -> getLastSent();
            sinfo -> lastReceived = ret -> getLastReceived();

            getHandleSet(
                        &sinfo -> service_handle, &sinfo -> outhandle,
                         &sinfo -> errhandle, &status);

            res -> totalsessions++;

            sinfo -> next = NULL;
            addnext = TRUE;
        }

        // through with iterator
        if (sessionIterator != NULL)
        {
            delete sessionIterator;
        }

    }

    UnlockSessionMutex();
}
```

```
/*********************************************************************
                                                            ********
**
** Routine:  removeSession
**
** Inputs:
**
** Outputs:
**
** Return Codes:
**            None
**
** Purpose: Remove the active session object between the GUI and the
                                                           Service.
**
**********************************************************************
                                                            ********
*/

int
removeSession(IN DD_client_session_id *sess_id,
              OUT int *status)
{
    EDMSession *sess;
    EDMSession *ret;

    if (status == NULL)
    {
        return -1;
    }

    if (sess_id == NULL)
    {
        *status = SESSION_BAD_ARGS;
        return -1;
    }

    *status = 0;
    if (G_sessionTree.isEmpty())
    {
        EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_LIST_EMPTY, 0,
                            "No sessions in list.
                                        Can't remove session <%ld:%ld>",
                            sess_id -> high, sess_id -> low);
        *status = SESSION_LIST_EMPTY;
        return -1;
    }

    sess = new EDMSession();

    if (sess == NULL)
    {
        EDMDispatch_logent(
                     __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                            "Failure to create a session block");
        *status = SESSION_NO_MEMORY;
        return -1;
    }

    sess -> setSessionID(sess_id);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.remove(sess);
```

```
    UnlockSessionMutex();

    if (ret == NULL)
    {
        EDMDispatch_logent(
                    __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                               "Failure to remove session %ld:%ld",
                               sess_id -> high, sess_id -> low);
        delete sess;
        *status = SESSION_LOOKUP_FAILED;
        return -1;
    }

    delete ret;
    delete sess;

    return 0;
}
```

D11

```
/**********************************************************************
                                                                 ******
**
** File Name:   RSLstart.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**       The intent of the contents of this file is to implement the
**       functions the control execution of the restore for the
                                                     Restore Service
**       Library.
**
**       These functions are provided to allow:
**      - creation of submit objects,
                                 which define the set of objects to be
**        restored and the scripts to be run before and after
                                                        restoration,
**      - starting the restoral of a submit object.
**
** The following functions comprise restoral management:
**
**      RSTSL_Start
**
**
** Compile-Time Options:
**            This section must list any compile time definitions
**            which will affect this header.
**
***********************************************************************
                                                               *****/


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/wait.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>


/*
 * Local headers
 */
```

```
#include <RSLinterns.h>
#include <RSLauxSupp.h>
#include <restore/EDMRESubmitApi.h>

#include <restore/REprogmsg.h>
#include <restore/dispatch_daemon.h>
#include <restore/EDMREProgressApi.h>

extern int RunExecutable(const boolean_ty ResetUid,
                         const int RunUid,
                         const char *starting_cwd,
                         const char *executable_name,
                         char **executable_argv,
                         char **executable_env,
                         int *run_exit_status,
                         boolean_ty *run_cancelled,
                         boolean_ty (*QuitTest)(void));

extern int RunWorkItemRestores(int, boolean_ty (*CancelRestoreTest)());

static eerrno_ty
ExecuteWorkItemRestore(int SubmitObjectID,
                       boolean_ty (*QuitTest)(void));

static eerrno_ty
RunPrepareRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int *PrepareExit);
static eerrno_ty
RunCleanupRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int runphase_status,
                  int *CleanupExit);

/*
 * #defines, structures, typedefs local to this source file
 */

#define STR_SURE(str) (str) ? str:""
#define REMOVE_NEWLINE(str)\
{\
   int rem_nl_index;\
   for(rem_nl_index = 0; str[rem_nl_index] != '\0'; rem_nl_index++)\
     {\
       if(str[rem_nl_index] == '\n')\
          str[rem_nl_index] = '\0';\
     }\
}

/*********************************************************************
 * Start
 *
 * This function begins execution of the restoral of the objects in a
 * submit object.  Its progress and requests for operator input are
 * returned via RSTSL_GetRestoreFeedback.
 *
 * Parameters:
 *
 * SubmitObjectID (
                I) - ID of the submit object which describes the restore
 * QuitTest       (I) - function to call to check for quit signal
 *
 ********************************************************************/
eerrno_ty RSTSL_Start( int SubmitObjectID, boolean_ty (*QuitTest)(
                                                                void))
{
    int ret_pre;
    int ret_exec;
    int ret_post;
    int ret_all_ok;
    int PrepareExit = 0;
    int CleanupExit = 0;
    boolean_ty QuitFlag = FALSE;

    char asctime[32];

    memset(asctime, 0, 32);

    (void)time(&rcp->rc_cmd_starttime);

    (void)ctime_r(&rcp->rc_cmd_starttime, asctime, 32);

    rcp->rc_cmd_last_waf_time = rcp->rc_cmd_starttime;

    REMOVE_NEWLINE(asctime);

    rbe_log_stats(0, "Restore Started at %s.", asctime);

    rbe_log_stats(0, "Restore Started application type %s.",
                  (rcp->rc_backup_app == 0)
                         ? "Network"
                         : ((struct pluginIDdata *)(
                                      rcp-> currentPIptr-> idData))
                                                              ->
                                                          name );

    rbe_log_stats(0, "Restore Started of "\
                  "top level object: %s. template %s, trailset %s.",
                  STR_SURE(rcp->rc_top_level_object_name),
                  STR_SURE(rcp->rc_template_name),
                  (rcp->rc_saveset_thread) ? "Alternate": "Primary");


    rbe_log_stats(0, "Restore Started by user %s, Uid %d, Gid %d.",
                  STR_SURE(rcp->rc_human_uidname),
                  rcp->rc_human_uid, rcp->rc_human_gids[0]);

    rbe_log_stats(0, "Restore Started with client destination %s.",
                  STR_SURE(rcp->rc_client_hostname));

    /* if not a network restore,
                           check if plugin has its own start function */

    if ( rcp-> rc_backup_app != 0
      && NULL != rcp-> currentPIptr-> piFuncArray[
                                       PIFuncIndexStartRestore ] )
    {
      setGlobalStatus( EDMRE_STATE_EXECUTE );
                                       /* set RE's internal status */
      ret_exec = rcp-> currentPIptr-> piFuncArray[
                                         PIFuncIndexStartRestore ]
                                         (
                                     rcp, SubmitObjectID, QuitTest);

      if( QuitTest() )      /* check for abort before return */
      {
        rbe_log_stats( EP_RB_RECOVER_ABORT,
                       "The restore was quit by the user during
                                                        execution.");
        setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                       /* set RE's internal status */
```

```
        return EP_RB_RECOVER_ABORT;
      }

      if (E_SUCCESS != ret_exec)
        setGlobalStatus( EDMRE_STATE_FAILED );
                                       /* set RE's internal status */
      else
        setGlobalStatus( EDMRE_STATE_SUCCESSFUL );
                                       /* set RE's internal status */

      return( ret_exec );
    }

    ret_pre = RunPrepareRestore(SubmitObjectID,
                                QuitTest,
                                &PrepareExit);

    if(EP_RB_RECOVER_ABORT == ret_pre)
    {
      rbe_log_stats(EP_RB_RECOVER_ABORT,
                    "The restore was quit by the user during
                                                    preparation.");
      setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                       /* set RE's internal status */
      return EP_RB_RECOVER_ABORT;
    }
    if(PrepareExit != 0)
    {
      rbe_log_stats(EP_RB_RECOVER_PREFAILED,
                    "The restore failed during preparation. Exit %d",
                    PrepareExit);

      setGlobalStatus( EDMRE_STATE_FAILED );
                                       /* set RE's internal status */
      return (EP_RB_RECOVER_PREFAILED);
    }

    setGlobalStatus( EDMRE_STATE_EXECUTE );
                                       /* set RE's internal status */

    ret_exec = ExecuteWorkItemRestore(SubmitObjectID,
                                      QuitTest);

    QuitFlag = QuitTest();          /* check for abort before cleanup */


    if (E_SUCCESS == (
              ret_all_ok = ret_exec) )      /* check if any WIs failed */
    {
      int local_stat;
      EDMStats stats;
      memset( &stats, 0, sizeof(EDMStats) );

      if (0 != getRestoreStatus( 0, &stats, &local_stat ))
      {
        rbe_log_stats(
            local_stat, "Internal error: Failed in getRestoreStatus.");
        ret_exec = ret_all_ok = EP_RB_RECOVER_EXECUTEFAILED;
      }
      else if (stats.edm.failed)
      {     /* if any workitems failed,
                                its a failure for cleanup purposes */
          if (0 == stats.edm.successful)
            ret_all_ok = EP_RB_RECOVER_ALLFAIL;
          else if (stats.edm.successful > stats.edm.failed)
            ret_all_ok = EP_RB_RECOVER_FEWFAIL;
```

```
         else
            ret_all_ok = EP_RB_RECOVER_MANYFAIL;
      }
   }

   ret_post = RunCleanupRestore(SubmitObjectID,
                                QuitTest,
                                ret_all_ok,
                                &CleanupExit);

   if(QuitFlag)
   {
     rbe_log_stats(EP_RB_RECOVER_ABORT,
                   "The restore was quit by the user during execution.
                                                                      ");
     setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                          /* set RE's internal status */
     return EP_RB_RECOVER_ABORT;
   }

   if (E_SUCCESS != ret_exec)      /* return execute status if it failed
                                                                      */
   {
     setGlobalStatus( EDMRE_STATE_FAILED );
                                          /* set RE's internal status */
     return ret_exec;
   }

   if(EP_RB_RECOVER_ABORT == ret_post)
   {
     rbe_log_stats(EP_RB_RECOVER_ABORT,
                   "The restore was quit by the user during cleanup.");
     setGlobalStatus( EDMRE_STATE_USER_QUIT );
                                          /* set RE's internal status */
     return EP_RB_RECOVER_ABORT;
   }

   if( (CleanupExit != 0) || (E_SUCCESS != ret_post) )
   {
     rbe_log_stats(EP_RB_RECOVER_POSTFAILED,
                   "The restore failed during cleanup. Exit %d",
                   CleanupExit);

     setGlobalStatus( EDMRE_STATE_FAILED );
                                          /* set RE's internal status */
     return (EP_RB_RECOVER_POSTFAILED);

   }

   setGlobalStatus( EDMRE_STATE_SUCCESSFUL );
                                          /* set RE's internal status */

   return( E_SUCCESS );

 }        /* RSTSL_Start */
```

```
static eerrno_ty
ExecuteWorkItemRestore(int SubmitObjectID,
                       boolean_ty (*QuitTest)(void))


{
    int ret_RunWItem;
    sm_push();

    rcp->error_message[0] = 0;

    if(0 != (ret_RunWItem = RunWorkItemRestores(
                                      SubmitObjectID, QuitTest)))
    {
      rbe_log_stats(0,"Internal error: Failed in RunWorkItemRestores");
    }

    sm_pop();

    if (QuitTest() == TRUE)
        return EP_RB_RECOVER_ABORT;

    if (ret_RunWItem != 0)
        return EP_RB_RECOVER_EXECUTEFAILED;

    return E_SUCCESS;
}
```

```
#define EXECUTABLE_MAX 1024
static eerrno_ty
RunPrepareRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int *PrepareExit)
{
  char **prephaseargs = NULL;
  char **prephaseenv = NULL;
  int GetSOstatus = 0;
  char preExecutable[EXECUTABLE_MAX];
  boolean_ty restore_cancelled = FALSE;

  *PrepareExit = 0;

  /*
   * GetSOPrePhase allocates prephaseargs & prephaseenv.
   * This will need to be free'ed later.
   *
   */

  if(0 != GetSOPrePhase(SubmitObjectID,
                        preExecutable,
                        EXECUTABLE_MAX,
                        &prephaseargs,
                        &prephaseenv,
                        &GetSOstatus))
  {
    rbe_log_stats(0,"Internal error: Failed in GetSOPrePhase");
    return (EP_RB_RECOVER_FATALERR);
  }

  if(0 != strcmp(preExecutable, ""))
  {
    setGlobalStatus( EDMRE_STATE_PREPHASE );
    if(-1 == RunExecutable(FALSE,
                           0,
                           NULL,
                           preExecutable,
                           prephaseargs,
                           prephaseenv,
                           PrepareExit,
                           &restore_cancelled,
                           QuitTest))
    {
      rbe_log_stats(
            0,"Internal error: Failed in RunExecutable for prepare.");
      return (EP_RB_RECOVER_FATALERR);
    }
    if(TRUE == restore_cancelled)
      return (EP_RB_RECOVER_ABORT);

  }

  return( E_SUCCESS );
}
```

```
boolean_ty alwaysFalse() { return FALSE; }
```

```
static eerrno_ty
RunCleanupRestore(int SubmitObjectID,
                  boolean_ty (*QuitTest)(void),
                  int runphase_status,
                  int *CleanupExit)
{

  char **postphaseargs = NULL;
  char **postphaseenv = NULL;
  int GetSOstatus = 0;
  char postExecutable[EXECUTABLE_MAX];
  boolean_ty restore_cancelled = FALSE;
  boolean_ty ignore_quit=FALSE;

  *CleanupExit = 0;

  /*
   * GetSOPostPhase allocates postphaseargs & postphaseenv.
   * This will need to be free'ed later.
   *
   */
  PurgeTrailQueue();
  if(0 != GetSOPostPhase(SubmitObjectID,
                         postExecutable,
                         EXECUTABLE_MAX,
                         &postphaseargs,
                         &postphaseenv,
                         &GetSOstatus))
  {
    rbe_log_stats(0,"Internal error: Failed in GetSOPostPhase");
    return (EP_RB_RECOVER_FATALERR);
  }

#define RESTORE_BREAK       "RESTORE_BREAK="
#define RESTORE_BREAK_TRUE  "RESTORE_BREAK=T"
#define RESTORE_BREAK_ERROR "RESTORE_BREAK=E"

  if(0 != strcmp(postExecutable, ""))
   {
     /*
      * If a quit has been specified, we need to tweak the
      * RESTORE_BREAK environment variable if set
      */
     char *abort=NULL;
     if(QuitTest())
       {
          abort=RESTORE_BREAK_TRUE;
       }
     else if(0!=runphase_status)
       {
          abort=RESTORE_BREAK_ERROR;
       }

     /*
      * ignore_quit is set to 1 when we have already processed a BREAK
      * (CANCEL from gui), and are using the environment variable
      * RESTORE_BREAK_E to signal the post restore script to clean
      * up form this break.  When that happens, an ignore_quit value
      * of 1 will cause actual quit signals to be ignores by the
      * cleanup script, since we already know (via the environment
      * variable) that we are in "cleanup mode" and no further signal
      * intercpetion is necessary.
      */
     ignore_quit=FALSE;
     if(NULL!=abort && NULL!=postphaseenv)
```

```
       {
          int isub=0;
          char *cptr;
          while(cptr=postphaseenv[isub])
            {
              if(strncmp(postphaseenv[isub],RESTORE_BREAK,strlen(
                                                        RESTORE_BREAK))==0)
                {
                  postphaseenv[isub]=esl_strdup(abort);
                  ignore_quit=TRUE;
                  if(NULL==postphaseenv[isub])
                    {
                      rbe_log_stats(
          EP_RB_RECOVER_MALLOC_FAILURE,"Allocate failed in RSLstart.c");
                      return EP_RB_RECOVER_POSTFAILED;
                    }
                }
              isub++;
            }
       }

     setGlobalStatus( EDMRE_STATE_POSTPHASE );
                                            /* set RE's internal status */
     if(-1 == RunExecutable(FALSE,
                            0,
                            NULL,
                            postExecutable,
                            postphaseargs,
                            postphaseenv,
                            CleanupExit,
                            &restore_cancelled,
                            ignore_quit?alwaysFalse:QuitTest))
     {
       rbe_log_stats(
             0,"Internal error: Failed in RunExecutable for cleanup.");
       return (EP_RB_RECOVER_FATALERR);
     }
     if(TRUE == restore_cancelled)
       return (EP_RB_RECOVER_ABORT);

   }
  return( E_SUCCESS );
}
```

```
static eerrno_ty
RunExecutionOverrideRestore(int SubmitObjectID,
                            boolean_ty (*QuitTest)(void))
{

  return( E_SUCCESS );
}
```

```
#undef EXECUTABLE_MAX
```

```
/*************************************************************************
                                                                   ******
**
** File Name:   RSLwisvr.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**      The intent of the contents of this file is to implement the
**      functions the control execution of work item restores for
                                                                 Restore
**      Service  Library.
**
**      These functions are provided to allow:
**
** The following functions comprise restoral management:
**
**      RunWorkItemRestores()
**
**
** Compile-Time Options:
**              This section must list any compile time definitions
**              which will affect this header.
**
**************************************************************************
                                                                  *****/


#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <values.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/ebutil.h>
#include <restore/REprogmsg.h>

/*
 * Local headers
 */

#include <RSLspexits.h>
#include <RSLremfd.h>
#include <EDMRESchedApi.h>
#include <EDMREHandleMgrApi.h>
#include <RSLinterns.h>
#include <RSLrbrmain.h>
#include <restore/EDMRESubmitApi.h>
#include <EDMREDrainApi.h>

#define STR_SURE(str) (str) ? str:""


/*
```

```
 *
 * RunWorkItemRestores
 * {
 * Set the number of drives being used for the life of the restore.
 * SetQuitFlag = FALSE
 * TrailRestoresLeft = { # of trail restores }
 * TrailRestoresRunning = 0;
 *
 *   while drive available.
 *     RunTrail -- Set drive concurrency for trail restore.
 *     TrailRestoresRunning++;
 *     while OKToRunWIForTrail
 *       StartWIRestore
 *
 *   while(1)
 *     if(QuitTest)
 *        Send WICancells
 *        SetQuitFlag = TRUE
 *     Select(from_pipe, timeout (5 seconds))
 *     for each WI that completes
 *       interperate return.
 *       Drain progress.
 *       Send final Progress for work item.
 *       if(WIfailed)
 *          if(OKtoReschedule && SetQuitFlag == FALSE)
 *             Add to TrailQueue
 *       if(TrailRestoreHasMoreWorkitems && SetQuitFlag == FALSE)
 *          while OKToRunWIForTrail
 *             StartWIRestore
 *       else -- RunNewTrailRestore
 *          EndTrailRestore(prevTrailQueue)
 *          TrailRestoresRunning--;
 *          TrailRestoresLeft--;
 *          while (drive available &&
 *                 SetQuitFlag == FALSE &&
 *                 TrailRestoresLeft)
 *
 *              RunTrail -- Set drive concurrency for trail restore.
 *              TrailRestoresRunning++;
 *              while (
 *                    OKToRunWIForTrail && SetQuitFlag == FALSE)
 *                  StartWIRestore
 *              end while
 *          end while
 *       end else
 *    End for each WI completes
 *    if (((SetQuitFlag == TRUE) && (TrailRestoresRunning == 0))||
 *         TrailRestoresLeft == 0)
 *         return;  exit loop.
 *  end while(1)
 *
 * }
 *
 */

/* Stubs */
static int
InterpretWorkItemRestoreResults(wi_restore_results *results);

static int
test_fd(int fd);

static void
DebugLogFds(char *error_msg,
            fd_set *fds);
```

```
static int
DetermineGlobalDriveUse();

static int
test_fd_hup(int fd);

static int
FindTrailIDForWItem(int handle,
                    int *TrailID,
                    int *status);

static int
SendRunningWorkItemsQuit();

/* End Stubs */

static int
Select(int nfds,
       fd_set *readfds,
       fd_set *writefds,
       fd_set *exceptfds,
       struct timeval *timeout);


static int
HandleWorkItemRestoreResults(int FromFD,
                             int *TrailID,
                             wi_restore_results *results);
static int
RunWorkItemRestoresForTrail(const int TrailID,
                            const int CountDrivesAvailable,
                            boolean_ty (*CancelRestoreTest)(),
                            boolean_ty *QuitFlag,
                            int *CountDrivesInUse);


/*
 * RunWorkItemRestores()
 *
 *  Runs a set of work item restores.
 *
 * Args:
 *   SubmitObject
 *   CancelRestoreTest().
 *
 * Returns: int 0 for success.
 */
int
RunWorkItemRestores(int SubmitObject, boolean_ty (*CancelRestoreTest)(
                                                                      ))
{
  boolean_ty QuitFlag = FALSE;  /* Has the user requested a quit.*/
  boolean_ty SentQuit = FALSE;  /* Have we initiated the quit. */

  int TrailRestoresRunning = 0;
                              /* The number of trail restores running. */
  int TrailRestoresLeft;
                                 /* The number of trail restores left. */
  int TrailRestoresTotal;
                                /* The number of trail restores total. */

  int CountDrivesAvailable;     /* The count of drives available. */
  int CountDrivesInUse = 0;     /* The count of drives in use. */

  int temp_status;
  int HighestActiveTrail = 0;
                          /* The trail queues are ordered from 1 to n. */
```

```
                                   /* This is the highest trail running
                                                                      */


  if(debugmode)
  {
    (void)rbe_user_error(0,
                         "DEBUG: Running RunWorkItemRestores.");
  }
  /* GenerateTrailQueues()
   * Buckets the work items into trail queues.
                                             The trail queues are sorted
   * in the order which the restores should run.
   */

  if(0 != GenerateTrailQueues(SubmitObject,
                              &TrailRestoresTotal,
                              &temp_status))
  {
    (void)rbe_user_error(0,
                         "Internal error: Cannot generate trail
                                  queues, cannot continue.");

    return -1;
  }

  TrailRestoresLeft = TrailRestoresTotal;

  CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);

  if(debugmode)
  {
    (void)rbe_user_error(0,
                         "DEBUG: RunWorkItemRestores for %d trails.",
                         TrailRestoresTotal);
  }
  /*
   * This is the start up loop to get the initial work item
   * restores started.
   */

  QuitFlag = CancelRestoreTest();

  while((CountDrivesInUse < CountDrivesAvailable) &&
        (HighestActiveTrail < TrailRestoresTotal) &&
        (FALSE == QuitFlag))
  {
    int submitObjID = 0;
    int submitelementID = 0;

    HighestActiveTrail++;

    /*
     * Activate the Trail Queue.
                         This allows the traill queues to be used to
     * determine the work item restores to run.
     */
    if(0 != ActivateTrailQueue(HighestActiveTrail,
                               1,
                               &temp_status))
    {
      (void)rbe_user_error(0,
                           "Internal error: Cannot activate trail
                    queues(1) for trailid %d, cannot continue.",
                      HighestActiveTrail);

      return -1;
    }
    /*
     * This sets the number of drives and media access concurrency for
                                                      trail restores.
     * i.e. The count of running work item restores for this trail.
                                                   Today this is one.
     */
    if(0 != SetTQDrivesAcquired(HighestActiveTrail, 1, &temp_status))
    {
      (void)rbe_user_error(0,
                           "Internal error: Cannot set drive acquired(
           1) for trailid %d, cannot continue.", HighestActiveTrail);
      return -1;
    }

    if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                                   HighestActiveTrail,

                                                 CountDrivesAvailable,

                                                    CancelRestoreTest,
                                                        &QuitFlag,

                                                  &CountDrivesInUse)))

    {

      /* RunWorkItemRestoresForTrail does its own error logging. */
      return -1;
    }

    if(temp_status == 0)
    {
      (void)rbe_log_stats(0,
                          "Trail %d restore had no work item to run(
                                        1).", HighestActiveTrail);
      /* more work may be need to recover from this error condition.
                                                                    */

    }
    if(temp_status > 0)
    {
      TrailRestoresRunning++;
    }
  } /* End while() initial startup loop */

  while(1)
  {
    int HighestFd = 0;
    fd_set WorkItemFromFds;
    int retStatus;
    struct timeval timeout = {5, 0};

    if((QuitFlag) && (!SentQuit))
    {
      (void)rbe_log_stats(0,
                          "Restore was quit by user. Quitting restore,
                                       this could take a while.");


      SendRunningWorkItemsQuit();
      SentQuit = TRUE;
    }

    if(0 != getFromSet(&WorkItemFromFds, &HighestFd, &retStatus))
    {
      (void)rbe_user_error(0,
                           "Internal error: Cannot get auxproc result
                                             fds, cannot continue.");
```

```
      return -1;
    }
#if 0
    if(debugmode)
    {
      DebugLogFds("The file descriptors to wait on are ",
                  &WorkItemFromFds);
    }
#endif
    if(0 > (retStatus = Select(HighestFd + 1,
                               &WorkItemFromFds,
                               NULL, NULL,
                               &timeout)))

    { /* error */
      (void)rbe_user_error(RBRECOVER_MKERR(errno),
                           "Internal error: Cannot get auxproc result
                                             fds, cannot continue.");


      return -1;
    }

    else if (0 == retStatus)
    { /* timed out */

      QuitFlag = CancelRestoreTest();

    }

    else
    { /* Available fds */

      int ReadyFds = retStatus;
      int FoundFds = 0;
      int index;

      if(debugmode)
      {
        DebugLogFds("The file descriptors ready to read are ",
                    &WorkItemFromFds);
      }

      /*
       * If there are available fds then we may want to
       * schedule the next work item restore. We should
       * check if the user initiated a quit.
       */
      QuitFlag = CancelRestoreTest();

      for(index = 0;
          (index < (HighestFd + 1)) && (FoundFds < ReadyFds);
          index++)
      {
        int StartWorkItemforTrail = 0;
        if(FD_ISSET(index, &WorkItemFromFds))
        {
          int TrailID;
          int TrailAcquired;
          wi_restore_results results;
          FoundFds++;

          memset(&results, 0, sizeof(wi_restore_results));

          if(0 != HandleWorkItemRestoreResults(index,
                                               &TrailID,
                                               &results))
          {
            /* HandleWorkItemRestoreResults will do its own logging!
                                                                  */
            return -1;
          }
          /*
           * This is where we may want to retry the Work Item
           * Based on if it passes or fails
           */

          CountDrivesInUse--;


          if (0 > (StartWorkItemforTrail =
                   RunWorkItemRestoresForTrail(TrailID,
                                               CountDrivesAvailable,
                                               CancelRestoreTest,
                                               &QuitFlag,
                                               &CountDrivesInUse)))
          {
            /* RunWorkItemRestoresForTrail does its own logging. */
            return -1;
          }

          else if (StartWorkItemforTrail == 0)
               /* 0 work items started aboved,
                * lets check to see if this is the last work item
                                                          left for
               * this trail
                */
          {
            int wiCount;

            if(0 != GetRunningWI(TrailID, &wiCount, &temp_status))
            {
              (void)rbe_user_error(0,
                                   "Internal error: Cannot determine
          number of running work items for trail, cannot continue.");

              return -1;
            }
            if(debugmode)
            {
              (void)rbe_user_error(0,
                                   "DEBUG: RunWorkItemRestores no
                           more work items left for trailid %d ,
                         but %d wiCount workitem still running.",
                                   TrailID, wiCount);
            }
            /* Testing for No work items left running or started
             * For this trail.
             */

            if((0 == wiCount) && (0 == StartWorkItemforTrail))
            {

              TrailRestoresRunning--;
              TrailRestoresLeft--;

              if(0 != DeactivateTrailQueue(TrailID, &temp_status))
              {
                (void)rbe_user_error(0,
                                     "Internal error: Cannot
    deactivate trail queue for trailid %d, cannot continue.", TrailID);

                return -1;
              }
```

```
              /* This test is to determine,
                                        if a trail restore finished,
               * there may be another not yet started trail,
                                                          If we have
               * drive available the next trail restore will be
                                                            started.
               */
              if ((0 != TrailRestoresLeft) &&
                  (HighestActiveTrail < TrailRestoresTotal) &&
                  (CountDrivesInUse < CountDrivesAvailable))
              {

                HighestActiveTrail++;

                if(0 != ActivateTrailQueue(HighestActiveTrail,
                                           1,
                                           &temp_status))
                {
                  (void)rbe_user_error(0,
                                       "Internal error: Cannot
        activate trail queue(2) for trailid %d, cannot continue.",
                                       HighestActiveTrail);
                  return -1;
                }

                if(0 != SetTQDrivesAcquired(
                            HighestActiveTrail, 1, &temp_status))
                {
                  (void)rbe_user_error(0,
                                       "Internal error: Cannot set
                drive acquired(2) for trailid %d, cannot continue.",
                                       HighestActiveTrail);
                  return -1;
                }

                if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                                  HighestActiveTrail,

                                                CountDrivesAvailable,

                                                   CancelRestoreTest,

                                                           &QuitFlag,

                                                 &CountDrivesInUse)))
                {
                  /* RunWorkItemRestoresForTrail does its own logging.
                                                                    */
                  return -1;
                }
                if(temp_status == 0)
                {
                  /* If this Trail Had no work items we
                   * Should attempt to run the next trails
                   * work items here. This would be an internal
                   * error if a trail queue had no work item
                   * restores.
                   */
                  (void)rbe_log_stats(0,
                                      "Internal error: Trail %d
       restore had no work item to run(1).", HighestActiveTrail);
                  return -1;
                }
                if(temp_status > 0)
                {
                        /* IF at least on work item was started for this
                                                                      trail,
                         * then we have started a new trail.
                         */
                        TrailRestoresRunning++;
                      }
                    }

                  }
                }
              }
            } /* end for() */

          } /* else Available fds */

          /*
           * Terminate the loop if either
           * ----------------------------
           * 1) Sent the work items the quit AND
           *    No Trail restores a running.
           * OR
           * 2) No more Trail restores are left.
           */
          if(((0 == TrailRestoresRunning) && (SentQuit)) ||
             (0 == TrailRestoresLeft))
          {
            break;
          }

        }  /* end while(1) */
        if((0 == TrailRestoresRunning) && (SentQuit))
        {
          (void)rbe_log_stats(0,
                              "Restore was quit by user.
                                                 Work item restore quit.");
        }
        return 0;
      }
```

```
/* Functions needed
SendRunningWorkItemsQuit();
InterpretWorkItemRestoreResults();
*/

static int
Select(int nfds,
       fd_set *readfds,
       fd_set *writefds,
       fd_set *exceptfds,
       struct timeval *timeout)
{
  int retSelect;

  do
  {
    retSelect = select(nfds,
                       readfds,
                       writefds,
                       exceptfds,
                       timeout);

  } while ((-1 == retSelect) && (EINTR == errno));

  return retSelect;
}
```

```
eperrno
InitiateWorkItemRestore(const int SubmitObjID,
                        const int SubmitElemID)
{
  struct auxproc AuxprocVitals;
  eerrno_ty StartupAPResults = EXIT_FAILURE;
  time_t StartTime;
  int TempStatus;
  char junk_executable[1024];
  char **junk_argv;
  char **AP_env = NULL;
  int SOstatus;
  char clientName[256] = "";
  int clientPort;
  int status;
  time_t EndTime;
  /* Lets see if there are any environment variables to set.
   *  The restore of the output variables are ingnored.
   */
  if(E_SUCCESS != GetSOExecutionPhase(SubmitObjID,
                               junk_executable, 1024,
                               &junk_argv,
                               &AP_env,
                               &SOstatus))
  {
    (void)rbe_user_error(0,
                         "Internal Error: Could not get environment
                                                      variables.");
    return -1;
  }

  if (E_SUCCESS == GetSERcmdConnect(SubmitObjID,
                              SubmitElemID,
                              clientName, 256,
                              & clientPort,
                              &SOstatus))
  {
   StartupAPResults = StartupAuxprocess(0 /* XXX */,
                                   &AuxprocVitals,
                                   AP_env,
                                   clientName,
                                   clientPort);
  }
  else
  {
    (void)rbe_user_error(0,
                    "Internal Error: Could not get Remote Client name
                                                                    &"
                    "port to connect.");
    return -1;
  }

  if(E_SUCCESS != StartupAPResults)
  {
    /* StartupAuxprocess does its own logging. */
    return -1;
  }

  /*
   * We need to close the bulk fd. This file descriptor
   * is not being used any more. If we do not close it
   * here we will have a file descriptor leak because
   * we won't be able to determine what it was when the
   * work item completes.
```

```
   */

  close(AuxprocVitals.xp_fd_bulk_to_x);

  time(&StartTime);

  if(0 != newHandleSet(AuxprocVitals.xp_fd_to_x,
                       AuxprocVitals.xp_fd_from_x,
                       AuxprocVitals.xp_fd_prog_from_x,
                       SubmitObjID,
                       SubmitElemID,
                       AuxprocVitals.xp_pid,
                       StartTime,
                       &TempStatus))
  {
    (void)rbe_user_error(
                 0, "Internal Error: Could not register handle set.");
    return -1;
  }


  if(0 > StartWorkItemRestore(rcp,
                              &AuxprocVitals,
                              SubmitObjID,
                              SubmitElemID))
  {
    /*
     * StartWorkItemRestore does logging if initialization fails
     */
    (void)rbe_user_error(
                0, "Error in StartWorkItemRestore  SubmitObjID %d,"
                   "SubmitElemID %d ",SubmitObjID,
                                           SubmitElemID);
    /*
     * the following code kills auxproc when recx or xcpio do not
                                                                start
     * we do not want an auproc sitting around.
     * if errors occur in deleteHandleSet or KillWorkItemRestore the
                                                          messages are
     * logged in those calls, plus,
                        we already know there was an error and that
     * is why we are doing this right now.
     */
    time(&EndTime);

    deleteHandleSet(
  AuxprocVitals.xp_fd_from_x, EndTime, EP_RB_RECOVER_ALLFAIL, &status);

    KillWorkItemRestore(
                     AuxprocVitals.xp_pid, AuxprocVitals.xp_fd_to_x);

    return -1;
  }

  return 0;

} /* InitiateWorkItemRestore() */
```

```c
/*
 *       interperate return.
 *       Drain progress.
 *       Send final Progress for work item.
 *       Delete the handle set.
 */

static int
HandleWorkItemRestoreResults(int FromFD,
                             int *TrailID,
                             wi_restore_results *results)
{
  int ret = 0;
  int retries = 0;
  int GetAuxprocResultsStatus;
  int TempStatus;
  int DrainResult;
  int DrainedFD;
  int wiCount;
  int AuxProcPid;
  int ToFD, getFromFD, ProgressFD;
  time_t EndTime;
  unsigned long jobstat;
  int timeout = 3; /* Lets try 3 seconds */
  boolean_ty fromFDHangUp = FALSE;

  ToFD = getFromFD = ProgressFD = -1;

  while(! (fromFDHangUp))
  {
    GetAuxprocResultsStatus = GetAuxprocResults(FromFD, results);

    if(-1 == GetAuxprocResultsStatus)
    {
      /* GetAuxprocResults() does its own logging */

     (void)rbe_user_error(0," Error in GetAuxprocResults");
      return -1;
    }
    if(0 == GetAuxprocResultsStatus)
    {
        if(test_fd_hup(FromFD) == 1)
          {
            fromFDHangUp = TRUE;
          }
    }

    /* The remote result are not always going to
     * be set. For example if the remote command
     * is not started correctly.
     */

    if(results->local_exit_set == TRUE)
    {
      break;
    }
    else
    {
      sleep(1);
      test_fd(FromFD);
      continue;
    }
  }
```

```c
  time(&EndTime);

  if(0 != PushDrainRequest(FromFD, &TempStatus))
  {
    (void)rbe_user_error(0,
                         "Internal error: Could not push drain
                                 request, cannot continue.");

    return -1;
  }
  /* Lets give the progress thread a chance to drain keeping busy in
   * the meanwhile.
   */

  if(0 != FindTrailQueueOfWI(FromFD, TrailID, &TempStatus))
  {
    (void)rbe_user_error(0,
                         "Internal error: Could not find trail id for
                              finished work item, cannot continue.");

    return -1;
  }

  if(0 != DecrementRunningWI(*TrailID, &wiCount, &TempStatus))
  {
    (void)rbe_user_error(0,
                         "Internal error: Could not decrement running
                            work items for trail, cannot continue.");

    return -1;
  }

  if(0 != getPID(FromFD, &AuxProcPid, &TempStatus))
  {
    (void)rbe_user_error(0,
                         "Internal error: Could not get auxproc pid
                                for work item, cannot continue.");

    return -1;
  }

  if(0 != getHandleSet(
                 FromFD, &ToFD, &getFromFD, &ProgressFD, &TempStatus))
  {
    (void)rbe_user_error(0,
                         "Internal error: Could not get auxproc file
                        descriptors for work item, cannot continue.");

    return -1;
  }
  if(FromFD != getFromFD)
  {
    (void)rbe_user_error(0,
                         "Internal error:  mismatch on from file
                        descriptors for work item, cannot continue.");
    return -1;
  }

  while (0 != (ret = PopDrainResult(timeout,
                            &DrainResult,
                            &DrainedFD,
                            &TempStatus)) && retries < 3)
      retries++;

  if (ret != 0)
  {
    (void)rbe_user_error(0,
                         "Internal error: Could not pop drain results,
                                                     cannot continue.");

    return -1;
  }

  /*
   *      Send final Progress for work item. XXX
   */

  /* Translate the local and remote error statuses
   * to an eperrno value:
   */

  if(0 != results->local_exit_status)    /* use local error, if any */
    switch (results->local_exit_status)
    {
    case XG_EXIT_ALLFAIL:
      jobstat = EP_RB_RECOVER_ALLFAIL;
      break;
    case XG_EXIT_MANYFAIL:
      jobstat = EP_RB_RECOVER_MANYFAIL;
      break;
    case XG_EXIT_FEWFAIL:
      jobstat = EP_RB_RECOVER_FEWFAIL;
      break;
    case SPEXIT_REMOTE_STDERR_PROTOCOL:
    case SPEXIT_REMOTE_STDERR_FAIL:
      jobstat = EP_RB_RECOVER_CLIENT_STDERR_FAIL;
      break;
    case XG_EXIT_STOPPED:        /* treat like signal */
    default:     /* check for signal termination vs all generic
                                                         failures */
      if ( XG_EXIT_SIGBASE < results->local_exit_status
        || XG_EXIT_STOPPED == results->local_exit_status )
      { /* killed by signal or stopped: separate error for sigpipe */
        if (XG_EXIT_SIGBASE + SIGPIPE == results->local_exit_status)
          jobstat = EP_RB_RECOVER_SERVER_SIGPIPE;
        else
          jobstat = EP_RB_RECOVER_SERVER_SIGNAL;
      }
      else
      { /* generic server failure, unless client failed too */
        jobstat = EP_RB_RECOVER_SERVERFAIL;
        if (0 != results->remote_exit_status)
          jobstat = EP_RB_RECOVER_BOTHFAIL;
      }
    }
  else if(0 != results->remote_exit_status)
    jobstat = EP_RB_RECOVER_CLIENTFAIL;
  else
    jobstat = E_SUCCESS;

  if( (0 != results->remote_exit_status) ||
      (0 != results->local_exit_status ) )
  {
   int  status=0;
   int  rc=0;
   char *templateName=NULL;
   char *winame=NULL;
   char *trailsetName=NULL;

   rc = getHandleSetInformation(FromFD,
                                &templateName,
                                &winame,
```

```
                                &trailsetName,
                                &status);
   rbe_log_stats(0, "Restore Failure of "\
                    "top level object: %s, template %s.",
                    STR_SURE(winame),
                    STR_SURE(templateName));
   free(templateName);
   free(winame);
   free(trailsetName);
  }

  if( 0 != deleteHandleSet(FromFD, EndTime, jobstat, &TempStatus))
  {
       (void)rbe_user_error(0,
                            "Internal error: Could not delete Handle
                                               Set, cannot continue.");

    return -1;
  }

  if(0 != KillWorkItemRestore(AuxProcPid,
                              -1 /* Hack this arg is not needed yet
                                                          cmd_to */))
  {
    (void)rbe_user_error(0,
                         "Internal error: Could not kill finished
                                    auxproc, cannot continue.");
    return -1;
  }

  close(ToFD);
  close(FromFD);
  close(ProgressFD);

  if(debugmode)
  {
    (void)rbe_user_error(0,
                         "DEBUG: HandleWorkItemRestoreResults Auxproc(
         PID %d) just finished for trailid %d work items left = %d.",
                         AuxProcPid,
                         *TrailID,
                         wiCount);

    (void)rbe_user_error(0,
                         "DEBUG: HandleWorkItemRestoreResults Auxproc(
          PID %d) results are local: %d: setP:%s remote: %d set:%s.",
                         AuxProcPid,
                         results -> local_exit_status,
                         (
                       results -> local_exit_set) ? "TRUE": "FALSE" ,
                         results -> remote_exit_status,
                         (
                      results -> remote_exit_set) ? "TRUE": "FALSE");


  }
  return 0;
} /* End HandleWorkItemRestoreResults() */
```

```
/*
 * RunWorkItemRestoresForTrail()
 *
 * Description
 *  This function starts all the work item for the
 *  trail. For no this is set to one but concurrency
 *  will can be supportted.
 *
 * Args:
 *  (I) TrailID -- The id for this trail.
 *  (I) CountDrivesAvailable -- the total drives available to restore.
 *  (O) QuitFlag -- indicatate whether the user has quit the restore.
 *  (O) CountDrivesInUse -- The count of trails in use by restore.
 *
 * Return int
 *  if -1 then an error has occurred.
 *  if 0 or greater then the number of trail restores started will be
 *  returned.
 *
 */
static int
RunWorkItemRestoresForTrail(const int TrailID,
                            const int CountDrivesAvailable,
                            boolean_ty (*CancelRestoreTest)(),
                            boolean_ty *QuitFlag,
                            int *CountDrivesInUse)
{
  int DriveAcquiredForTrail;
  int DriveConcurencyForTrail;
  int submitObjID;
  int submitelementID;
  int popResults = 0;
  int temp_status;
  int CountOfWorkItemRestoreStarted = 0;
  int wiCount;

  while(1)
  {


    (*CountDrivesInUse)++;

    if((0 != (popResults = PopWIFromTrailQueue(TrailID,
                                               &submitObjID,
                                               &submitelementID,
                                               &temp_status))) &&
       (SCHED_NO_MORE_JOBS != temp_status))
    {
      (void)rbe_user_error(0,
                          "Internal error: Cannot pop work item off
                                    trail queue, cannot continue.");
      return -1;
    }

    if((-1 == popResults) && (SCHED_NO_MORE_JOBS == temp_status))
    {
      return CountOfWorkItemRestoreStarted;
    }

    temp_status = InitiateWorkItemRestore(
                                         submitObjID, submitelementID);

    if(temp_status != 0)
```

```
    {
      /* InitiateWorkItemRestore() does its own logging */
      (void)rbe_user_error(0, "Error in InitiateWorkItemRestore,"
           "submitObjID  %d, submitelementID %d", submitObjID,
                                                   submitelementID);
      return -1;
    }

    if(0 != IncrementRunningWI(TrailID, &wiCount, &temp_status))
    {
      (void)rbe_user_error(0,
                          "Internal error: Could not increment
                   running work items for trail, cannot continue.");
      return -1;
    }

    CountOfWorkItemRestoreStarted++;

    if(0 != GetTQDrivesAcquired(TrailID,
                               &DriveAcquiredForTrail,
                               &temp_status))
    {
      (void)rbe_user_error(0,
                          "Internal error: Cannot get drives
                                 acquired, cannot continue.");

      return -1;
    }

    if(0 != GetTQDriveConcurrency(TrailID,
                                 &DriveConcurencyForTrail,
                                 &temp_status))
    {
      (void)rbe_user_error(0,
                          "Internal error: Cannot get drive
                              concurrency, cannot continue.");
      return -1;
    }

    *QuitFlag = CancelRestoreTest();

    if((DriveAcquiredForTrail < DriveConcurencyForTrail) &&
       ((*CountDrivesInUse) < CountDrivesAvailable) &&
       (FALSE == *QuitFlag))
    {
      continue;
    }
    else
    {
      break;
    }
  }
  return CountOfWorkItemRestoreStarted;
} /* RunWorkItemRestoresForTrail() */
```

```
/* Stub */
static int DetermineGlobalDriveUse()
{
/* Limiting to MAXINT === not limiting... Need resource management
 * to do this properly.
                   NOTE: This should now work like eb_dc_restore does.
 */
  return MAXINT;
}
```

```
static int
SendRunningWorkItemsQuit()
{
  int *APlist;
  int count;
  int status;
  int index;

  if(0 != getPIDList(&count, &APlist, &status))
  {
    (void)rbe_user_error(0,
                         "Internal error: Cannot get auxproc pid list,
                                                 cannot continue.");
    return -1;
  }

  for(index = 0; index < count; index++)
  {
    QuitWorkItemRestore(APlist[index]);
  }
  return 0;
}
```

```
/*
 * Stub this out for now.
 */
static int
InterpretWorkItemRestoreResults(wi_restore_results *results)
{
  return 0;
}
```

```
static void
DebugLogFds(char *error_msg,
            fd_set *fds)
{
  int index, fd_count = 0;
  char buffer[4096];
  char *bufptr = (char*)buffer;


  for(index=0;
      index < 1024;
      index++)
  {
    if( FD_ISSET(index, fds))
    {
      int size = 0;
      size = sprintf(bufptr,"%d,",index);
      bufptr += size;
      fd_count++;
    }
  }
  rbe_log_stats(0,"%s fd_count:: %d :: {%s}\n",
                error_msg, fd_count, buffer);
}
```

```
static int
test_fd(int fd)
{
  fd_set read_fd;
  int ret_select;
  struct timeval timeout = {0, 0};

  FD_ZERO(&read_fd);

  FD_SET(fd, &read_fd);

  do
  {
    ret_select = select(fd + 1, &read_fd, NULL, NULL,  &timeout);

  } while((-1 == ret_select) && (EINTR == errno));

  return ret_select;

}
```

```
/*
 * test_fd_hup()
 *
 * Description: Test the supplied file descriptor to see if
 *   it has had the hang up condition.
 *
 * Args:
 *  Input fd -- the file descriptor to check for the hang up condition.
 *
 * Returns:
 *  1 for HUP event received on fd.
 *  0 No HUP event received on fd.
 *  -1 errno set.
 *
 */
static int
test_fd_hup(int fd)
{
  struct pollfd fds;
  int ret_poll;

  if(fd < 0)
  {
    errno = EINVAL;
    return -1;
  }

  fds.fd = fd;
  fds.events = POLLIN;
  fds.revents = 0;  /* initialize */

  do
  {
    ret_poll =  poll(&fds, 1, 0);

  } while((-1 == ret_poll) && (EINVAL == errno));

  if(-1 == ret_poll)
  {
    return -1;
  }

  if(POLLHUP & fds.revents)
  {
    return 1;
  }
  else
  {
    return 0;
  }

} /* end test_fd_hup() */
```

```
/**********************************************************************
                                                                 ******
**
** File Name:   RSLauxmgr.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
** -----------
**      The intent of the contents of this file is to implement the
**      functions the control execution of work item restores Or the
**      running of auxproc.
**
**
**      Library.
**
**      These functions are provided to allow:
**         - The pipe, fork, duping closing and exec of auxproc.
**         - The starting of work item restores.
**         - The quitting of work item restores.
**         - The getting results of work item restores.
**
**
** The following functions comprise restoral management:
**
**
**
**
**
** Compile-Time Options:
**                This section must list any compile time definitions
**                which will affect this header.
**
**********************************************************************
                                                                *****/


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */

#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebutil.h>
#include <ebreport/ebvl.h>
```

```
/*
 * Local headers
 */

#include <RSLinterns.h>
#include <RSLrbrmain.h>
#include <restore/EDMRESubmitApi.h>
#include <EDMfork.h>
#include <RSLauxSupp.h>

#define SUBMIT_FIELD_MAX 2048

extern int putenv(const char *string);

extern char *strsignal(int sig);

extern int ChildDone(int child_pid, int *child_result);

static void
reset_recovery_privileges(struct recover_context *rcx,
                          int changed);

static void
set_recovery_privileges(struct recover_context *rcx,
                        int *changed);

static char *
generate_rcmdpath(int SubmitObjectID, int SubmitElemID);

/*
 * StartupAuxprocess()
 *
 * Description:
 *  This function pipe, fork,  & exec auxproc. After which it tests
 *  its the just started auxproc with the Ping command.
                                                      Some additional
 *  processing is it closes the fds that are not associated with
                                                              auxproc
 *  but are inherited from the parent (
                                  restore engine). If any environment
 *  variable need to be set for auxproc, they are set.
 *
 * Args:
 * (Inp) debugmode -- int 1 for debug and 0 for no debug.
 * (Out) struct auxproc *xp -- preallocated structure to return vital
                                                                 info
 *                          for auxproc.
 * (char) **auxproc_envp -- environment variables to be appended to
                                                            auxproc's
 *                          environment.
                                     The rest of the environment is
 *                          inherited from auxproc.
 *                          Format is "ENV=value\0"
 * char  *socketClientNm  -- used for the client initiated restore,
                                                          this is the
 *                          client name
 * int  clientSocketPort -- Used to identify the port number on which
                                                                   to
 *                          contact the client
 * Notes:
 *     auxproc_envp can be NULL indicating no environment to append to
                                                             auxproc.
 *     This is a char ** which last char * should be NULL.
 *
 * Inheritted from:
```

```
 *  static eerrno_ty  setup_aux_processes(struct recover_context *rcx)
 */

eerrno_ty
StartupAuxprocess(int          debugmode,
                  struct        auxproc *xp,
                  char          **auxproc_envp,
                  char    *socketClientNm,
                  int     clientSocketPort)
{

  /*
   * NOTES::
   * 1) Do I really want to be reliant on the recover_context struct.
   * 2) I need to fork() exec() auxproc.
   * 3) Is fork1 really going to work.
   */


#define RFD       0        /* in a pipe, fd 0 is the read descriptor */
#define WFD       1        /* and fd 1 is the write descriptor */
    int save_errno;
    int fd;
    char *resultsbuf = NULL;
    char *Auxproc_pathname = AUXPROCPATHNAME;
    char *Auxproc_executable = AUXPROCNAME;

    int ping_status;
    int auxproc_index = 0; /* alton Remove */
    int cmd_pipe_to[2];
    int cmd_pipe_from[2];
    int bulk_pipe_to[2];
    int prog_pipe_from[2];

    if (pipe(cmd_pipe_to) == -1 ||
        pipe(cmd_pipe_from) == -1 ||
        pipe(bulk_pipe_to) == -1 ||
        pipe(prog_pipe_from) == -1)
    {
        save_errno = errno;
        rbe_log_stats(RBRECOVER_MKERR(errno), "pipe() failed");
        return(RBRECOVER_MKERR(save_errno));

    }
    /* This below appends environment variables to
     * the evironment that auxproc inherits from the
     * restore engine.
     */

    if(NULL != auxproc_envp)
      {
        int index;
        for(index=0; NULL != auxproc_envp[index]; index++)
          {
            if(0 != putenv(auxproc_envp[index]))
              {
                rbe_log_stats(RBRECOVER_MKERR(errno),
                              "Unable to set auxproc environment %s,
                                        for auxproc PID %d.",
                             auxproc_envp[index], getpid());

                _exit(1); /* We are the child */
              }
          }

        }
```

```
    switch (xp->xp_pid = EDMfork1())
    {
        case -1: /* Error */
           save_errno = errno;
           rbe_log_stats(RBRECOVER_MKERR(errno), "fork() failed");
           return(RBRECOVER_MKERR(save_errno));

         case 0: /* child */


            {
           char procnum_str[32];
           char r_fd_str[32];
           char w_fd_str[32];
           char r_bulk_fd_str[32];
           char w_prog_fd_str[32];
           char dbgmodestr[32];
          char socket_host_str[256];
                                                /* Not sure what this should be */
          char socket_port_str[32];
           int ret_exec = 0;
          socket_port_str[0] = '\0';

           (void)sprintf(procnum_str, "%d", auxproc_index);
           (void)sprintf(r_fd_str, "%d", cmd_pipe_to[RFD]);
           (void)sprintf(w_fd_str, "%d", cmd_pipe_from[WFD]);
           (void)sprintf(r_bulk_fd_str, "%d", bulk_pipe_to[RFD]);
           (void)sprintf(w_prog_fd_str, "%d", prog_pipe_from[WFD]);
           (void)sprintf(dbgmodestr, "%d", debugmode);
           if (NULL != socketClientNm)
             {
               (void)sprintf(socket_host_str, "%s", socketClientNm);
               (void)sprintf(socket_port_str, "%d", clientSocketPort);
             }
           else
             {
              socket_host_str[0] = 0;
              socket_port_str[0] = 0;

             }
           ret_exec = execlp(
                        Auxproc_pathname,            /* prog to execute */
                                Auxproc_executable,
                                                                /* argv 0 */
                                procnum_str,
                                r_fd_str,
                                w_fd_str,
                                r_bulk_fd_str,
                                w_prog_fd_str,
                                dbgmodestr,
                                socket_host_str,
                                socket_port_str,
                                (char *)0);

           rbe_log_stats(RBRECOVER_MKERR(errno),
                         "Unable to exec %s, for %s PID %d.",
                         AUXPROCNAME,
                         AUXPROCNAME,
                         getpid());


           _exit (1);
           }

    default: /* parent */
```

```
        /*
         * The parent has no need for the fd
         * used to write *to* the parent, nor
         * the fds used to read *from* the parent.
         * If these are not close EPIPE and SIGPIPE
         * may be missed by the writer when the
         * intended reader dies.
         */

        (void)close(cmd_pipe_to[RFD]);
        (void)close(bulk_pipe_to[RFD]);
        (void)close(cmd_pipe_from[WFD]);
        (void)close(prog_pipe_from[WFD]);

        /*
         * but does want to save the other fds
         */

        xp->xp_fd_from_x    = cmd_pipe_from[RFD];
        xp->xp_fd_to_x      = cmd_pipe_to[WFD];
        xp->xp_fd_bulk_to_x = bulk_pipe_to[WFD];
        xp->xp_fd_prog_from_x = prog_pipe_from[RFD];
        /*
         * In debugmode, exec the separate-process
         * version of the auxprocs (ptrace issue)
         */

        if (debugmode)
        {
          auxcmdpacket(xp->xp_fd_to_x, 'X', 0, "");
        }
    }
#define PING_TEST_STR "abc"
#define PING_TEST_STR_SIZE 4 /* include the '\0' */

    fd = xp->xp_fd_to_x;
    auxcmdpacket(fd, 'P', PING_TEST_STR_SIZE, PING_TEST_STR);

    fd = xp-> xp_fd_from_x;

    ping_status = auxresults(fd, 'P', 0, &resultsbuf);

    if ((-1 == ping_status) ||
        (NULL == resultsbuf) ||
        (strcmp(resultsbuf, PING_TEST_STR) != 0))
    {
      rec_api_log_csm(SUB_CSM_NO_PING_AUXPROC, NULL);
      rbe_log_stats(0, "%s ping start-up",
                    AUXPROCNAME);
      return(EP_RB_RECOVER_AUXPROC_DIED);
    }

    free(resultsbuf);

    return(E_SUCCESS);

#undef PING_TEST_STR
#undef PING_TEST_STR_SIZE
#undef RFD
#undef WFD
#undef MAX_FD
}        /* end of setup_aux_process() */
```

```
#define MAX_SUBMIT_FIELD 2048


/*********************************************************
 * start_cpiogen()
 *
 *  This function initiates a work item restore. It first determines
 *  the size of the restore command to send to auxproc, malloc's the
 *  memory and creates the restore command and sends it to auxproc.
 *  Auxproc will start the rcmd (if necessary) and start xcpiogen.
 *  Auxproc will send the initialization reply which is read by this
 *  function. The results are for initialization.
 *
 * Args:
 *  (I) rcx -- Recover Context struct (Limited use)
 *  (I) xp -- auxproc struct pointer.
 *  (I) submitObjectID -- idententifies what and how to run restore.
 *  (I) submitElemID --  idententifies what and how to run witem
                                                            restore.
 *  (I) auxprocnum -- auxoroc number may become obsolete.
 *  (I) *rcmdinfo[4] -- the rcmdinfo must be included.
 *  (O) results -- Of the work item restore initialization.
 *  (O) err_str -- Of work item initialization failures.
 *
 * Returns:
 *  xcpiogen's pid for success, -1 for failure.
 *
 * NOTES::
 *  rcx -- The recover context structure is carefully used below.
 *  The rcx structure should be used only to get the global values
 *  like the xcpiogen exectutable name and the config structure.
 *
 *  Submit Object should be used to determine user id,
 *  admin priveledges, and other values that would not vary
 *  for a potentially multi work item restore.
 *
 *  Submit Element  should be used to determine anything that
 *  could be potentially unique for a work item restore.
 *
 *  rcmdinfo is the command line for the remote command.
 *********************************************************/
int
start_cpiogen(struct recover_context *rcx,
              struct auxproc *xp,
              int submitObjectID,
              int submitElemID,
              int auxprocnum,
              char *rcmdinfo[4],
              rcmd_pkt0_info *results,
              char **err_str)
{
    char *auxproc_databuf;
    size_t data_len = 0;
    char *p;
    int i;
    int resfd;

    /* For Initialization results */
    int pid = -1;
    rcmd_pkt0_info *pkt0p;
    rcmd_pkt0_info pkt0p_buffer;
    char *resultsbufptr = NULL;
    char *errstr = "";
    struct mark_summary submit_summary;
    /* Args for xcpiogen */
    char *xcpiogen_argv0;
    int xcpiogen_argc;
    char *submit_file_flag = "-S";
    char *outputfd_fmt = "-f%d";
    char *progress_report_flag = "-p";
    u_hyper total_bytes;              /* From the summary */
    char *total_bytes_flag = "-B";
    char *total_bytes_stringP;
    char *bufsizeP = NULL;
    char bufsize_buffer[20];

    RBC_WORKGROUP *pg;
    RBC_WORKITEM *pi;

    /* temporary variable for the submit object / element fields. */
    char temp_effective_uidname[MAX_SUBMIT_FIELD];
    char temp_socket_host[MAX_SUBMIT_FIELD];
    char temp_workitem_name[MAX_SUBMIT_FIELD];
    char temp_submit_file[MAX_SUBMIT_FIELD];
    int  temp_socket_port;
    int GetSEStatus = 0;
    int GetSOStatus = 0;

    if( GetSEWorkItemName(submitObjectID, submitElemID,
                          temp_workitem_name, MAX_SUBMIT_FIELD,
                          &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get work item name.");
      return -1;
    }

    if( GetSOEffectiveUserName(submitObjectID,
                               temp_effective_uidname,
                               MAX_SUBMIT_FIELD, &GetSOStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get user name.");
      return -1;
    }


    if( GetSERcmdConnect(submitObjectID, submitElemID,
                         temp_socket_host,  MAX_SUBMIT_FIELD,
                         &temp_socket_port, &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get socket host/port pair.");
      return -1;
    }

    if( GetSESubmitFile(submitObjectID, submitElemID,
                        temp_submit_file,  MAX_SUBMIT_FIELD,
                        &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get submit file.");
      return -1;
    }


    /*
     * +1 for '\0' characters
     */

    data_len += strlen(rcmdinfo[0]) + 1;          /* rcmd-hostname */
    data_len += strlen(rcmdinfo[1]) + 1;          /* rcmd-locuser */
    data_len += strlen(rcmdinfo[2]) + 1;          /* rcmd-remuser */
```

```
    data_len += strlen(rcmdinfo[3]) + 1;          /* rcmd-cmd */

    data_len += strlen(temp_effective_uidname) + 1;

    data_len += sizeof (int);                          /* fd info */
    data_len += sizeof (int);                          /* flags */
    data_len += strlen(rcx -> rc_cpiogen_executable) + 1;
                                                      /* xcpiogen-cmd */
    data_len += sizeof (int);
                                                     /* xcpiogen-argc */

    /*
     * xcpiogen arguments.  Pass traditional argv[0]
     * plus the output file descriptor number,
     * plus -p (progress report mode) and -B with its
     * argument (total bytes to be processed).
     */

    xcpiogen_argc = 7;

    xcpiogen_argv0 = strrchr(rcx -> rc_cpiogen_executable, '/');
    if (NULL != xcpiogen_argv0)
    {
        ++xcpiogen_argv0;
    }
    else
    {
        xcpiogen_argv0 = rcx -> rc_cpiogen_executable;
    }
    data_len += strlen(xcpiogen_argv0) + 1;
    data_len += strlen(submit_file_flag) + 1;
                                            /* xcpiogen argv[1] */
    data_len += strlen(temp_submit_file) + 1;
                                            /* xcpiogen argv[2] */
    data_len += strlen(outputfd_fmt) + 1;
                                            /* xcpiogen argv[3] */
    data_len += strlen(progress_report_flag) + 1;
                                            /* xcpiogen argv[4] */

    /*
     * Get the current total number of bytes to be processed from
     * the mark summary u_hyper so that we can then convert it to
     * a decimal string to be passed to xcpiogen().
     */

    if(0 != GetSEMarkedSummary(submitObjectID,
                               submitElemID,
                               &submit_summary,
                               &GetSEStatus))
    {
      /* This is not a critical error. This may cause progress
       * reporting problems!
       */

      rbe_log_stats(0, "Unable to set submit size for xcpiogen.");
      total_bytes = ul_to_uh(0);
    }
    else
    {
      total_bytes = submit_summary.len_mkd_files;
    }

    total_bytes_stringP = u_hyper_to_decimal(total_bytes);

    /*
     * Add the size of the "total bytes" flag and value string
     */

    data_len += strlen(total_bytes_flag) + 1;
                                            /* xcpiogen argv[5] */
    data_len += strlen(total_bytes_stringP) + 1;
                                            /* xcpiogen argv[6] */

    /*
     * Add size for db API socket info
     */

    data_len += sizeof (int);                   /* socket port # */
    data_len += strlen(temp_socket_host) + 1;

    /*
     * locate work item in config info & get length of filespec
     * %%% changed break in inner loop to 'goto' to resume with
     *      found item.
     */

    for (pi = NULL, pg = rcx->rc_config->pgrouplist /* OK */;
         NULL != pg;
         pg = pg->next)
    {
        for (pi = pg->pwilist; NULL != pi; pi = pi->next)
        {
            if (0 == strcmp(pi->name, temp_workitem_name))
            {
                goto stopsearch;     /* %%% exit both loops */
            }
        }
    }

stopsearch:
    if (pi != NULL)
    {
        data_len += strlen(pi->list)+1;
    }
    else
    {
        data_len += 1;
    }

    if (NULL != pi && DEFAULT_EB_BUFSIZE !=
                                        pi->recover_server_bufsize)
    {
        sprintf(bufsize_buffer, "-R%d", pi->recover_server_bufsize);
        bufsizeP = bufsize_buffer;
        data_len += strlen(bufsizeP) + 1;
        xcpiogen_argc++;        /* one more arg to xcpiogen */
    }

    data_len += strlen(temp_workitem_name) + 1;

    /*
     * Allocate memory to hold all this gunk we need to
     * shove towards our auxilliary process.
     */

    auxproc_databuf = sm_fmalloc((unsigned)data_len);

    /*
     * Fill in the gunk.  First, the rcmd info for the rsh part.
     */

    p = auxproc_databuf;
```

```
    for (i = 0; i < 4; i++)
    {
        (void)strcpy(p, rcmdinfo[i]);
        p += strlen(p)+1;
    }

    /*
     * The human username which the remote should operate as
     */

    (void)strcpy(p, temp_effective_uidname);
    p += strlen(p)+1;

    /*
     * The xcpiogen-cmd-fd-info, which tells auxproc which
     * arg has the sprintf format string to insert the
     * actual output fd number. Remember auxproc sets up
     * the connections between xcpiogen and the rcmd. This
     * below arg is sent to auxproc to tell auxproc which
     * argument to update the format sting with the outputfd.
     */

    i = 3;              /* For argv[3] -- See outputfd_fmt below.*/

    memcpy(p, &i, sizeof(int));

    p += sizeof (int);

    /*
     * the flags, which are always zero currently
     */

    i = 0;
    memcpy(p, &i, sizeof (int));
    p += sizeof (int);

    /*
     * The "xcpiogen command" that we will run locally.
     */

    (void)strcpy(p, rcx -> rc_cpiogen_executable);
    p += strlen(p)+1;

    /*
     * The argc for the "xcpiogen command", and its argv vector.
     */

    memcpy(p, &xcpiogen_argc, sizeof (int));
    p += sizeof (int);

    /*
     * The argv vector, which is argv0, the outputfd
     * thingy, and the progress report mode flag.
     */

    (void)strcpy(p, xcpiogen_argv0);
    p += strlen(p)+1;

    (void)strcpy(p, submit_file_flag); /* xcpiogen argv[1] */
    p += strlen(submit_file_flag) + 1;

    (void)strcpy(p, temp_submit_file); /* xcpiogen argv[2] */
    p += strlen(temp_submit_file) + 1;

    (void)strcpy(p, outputfd_fmt); /* xcpiogen argv[3] */
    p += strlen(outputfd_fmt) + 1;
    (void)strcpy(p, progress_report_flag); /* xcpiogen argv[4] */
    p += strlen(progress_report_flag) + 1;

    if (bufsizeP != NULL)
    {
        (void)strcpy(p,bufsizeP); /* xcpiogen argv[??] */
        p += strlen(bufsizeP) + 1;
    }

    /*
     * Follow this with the total bytes flag and value.
     */

    strcpy(p, total_bytes_flag); /* xcpiogen argv[??] */
    p += strlen(total_bytes_flag) + 1;
    strcpy(p, total_bytes_stringP); /* xcpiogen argv[??] */
    p += strlen(total_bytes_stringP) + 1;

    /*
     * socket info for db API
     */

    (void)memcpy(p, (char *)&temp_socket_port, sizeof (int));
    p += sizeof (int);

    /*
     * socket host name for db API
     */

    (void)strcpy(p, temp_socket_host);
    p += strlen(p)+1;

    if (NULL != pi) /* send filespec */
    {
        (void)strcpy(p, pi->list);
        p += strlen(p)+1;
    }
    else
    {
        *p++ = 0;
    }

    /*
     * workitem name
     */

    (void)strcpy(p, temp_workitem_name);
    p += strlen(p)+1;

    /*
     * assert that our arithmetic above was done correctly
     */

    if ((size_t)(p - auxproc_databuf) != data_len)
    {
        rbe_log_stats(0, "assertion failed: cmd size miscount");
        return -1;
    }

    /*
     * Send the restore command to auxproc. Auxproc will start
     * The remote command (if necessary) and xcpiogen.
     */

    auxcmdpacket(xp-> xp_fd_to_x,
```

```
                     'r', (int)data_len, auxproc_databuf);

    /*
     * Obtain the fork status
     */

    resfd = xp-> xp_fd_from_x;

    i = auxresults(resfd, '0', 0, &resultsbufptr);

    if (i < 0)
    {
        rbe_log_stats (0,
                       "** Error while starting auxproc for work item
                                                              \"%s\"",
                       temp_workitem_name);
        null_free (resultsbufptr);
        return -1;
    }

    /* This memory management is crap */

    pkt0p = &pkt0p_buffer;
    memcpy(pkt0p, resultsbufptr, sizeof(pkt0p_buffer));
    memcpy(results, resultsbufptr, sizeof(pkt0p_buffer));

    if ((pkt0p->msglen) > 0)
    {
        errstr = resultsbufptr + sizeof *pkt0p;
        *err_str = esl_strdup(errstr);
    }
    else
    {
        errstr = "";
        *err_str = esl_strdup("");
    }

    /*
     * if the fork failed, the cpiogen start fails
     */

    if (0 != pkt0p->failcode)
    {
        int jnk;


        if (strlen(errstr) > (size_t)0)
        {
            rbe_log_stats(0,
                          "** Error while starting auxproc, error %s,
                                                                       "
                    "for work item \"%s\"" ,errstr,
                                      temp_workitem_name);
        }

        free (resultsbufptr);

        /*
         * collect the useless 'r' reply packet
         */

        resultsbufptr = (char *)&jnk;
        (void)auxresults(resfd, 'r', sizeof (int), &resultsbufptr);
        return -1;
    }
    /*
     * Caller assumes responsibility for (eventually)
     * collected exit status of remote and local programs.
     */

    pid = pktOp->pid;

    free (resultsbufptr);

    return pid;
}       /* end of start_cpiogen() */
```

```
/*
 * 1) Remove rcx references.
 */

static char *
make_remote_cpiogen_cmd(struct recover_context *rcx,
                        int SubmitObjectID,
                        int SubmitElemID)
{
    char *rcmdpath = generate_rcmdpath(SubmitObjectID,
                                       SubmitElemID);
    char *minus_c = "";
    char *minus_c_arg = "";
    char *noclobber = "";
    char *remdebugflag = "";
    char *cmd;
    unsigned len;
    RBC_WORKITEM *work_itemP;
    char *bufsizeP = "";
    char bufsize_buffer[20];

    char temp_dirtop[SUBMIT_FIELD_MAX];
    char temp_workitem_name[SUBMIT_FIELD_MAX];
    OverwritePolicy temp_overwrite_policy;
    int GetSEStatus = 0;

    if (NULL == rcmdpath)
    {
      return NULL;
    }
    temp_overwrite_policy = GetSEDestOverWritePolicy(SubmitObjectID,
                                                     SubmitElemID,
                                                     &GetSEStatus);

    if (0 != GetSEStatus)
    {
      rbe_log_stats(0, "Unable to get overwrite policy.");
      return NULL;
    }

    if (GetSEDestDirTop(SubmitObjectID, SubmitElemID,
                        temp_dirtop, SUBMIT_FIELD_MAX,
                        &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get dirtop.");
      return NULL;
    }

    if (GetSEWorkItemName(SubmitObjectID, SubmitElemID,
                          temp_workitem_name,
                          SUBMIT_FIELD_MAX,
                          &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get work item name.");
      return NULL;
    }

    if (temp_dirtop != NULL)
    {
        minus_c = " -c ";
        minus_c_arg = temp_dirtop;
    }

    work_itemP = rbc_find_workitem_in_config(temp_workitem_name,
                                        NULL, NULL,
                                        rcx->rc_config, /* OK */
                                   RBC_FIND_WORKITEM_NO_OPTIONS);
    if (work_itemP != NULL
         && DEFAULT_EB_BUFSIZE != work_itemP->recover_client_bufsize)
    {
         sprintf(bufsize_buffer, " -A -b -A %d",
                 work_itemP->recover_client_bufsize);
         bufsizeP = bufsize_buffer;
    }

    switch (temp_overwrite_policy)
    {
    case RC_OVERPOL_NO_CLOBBER:
         noclobber = " -A -onever";
         break;

    case RC_OVERPOL_NEW_CLOBBER:
         noclobber = " -A -onewer";
         break;

    default:
         break;  /* do something better here */
    }

    if (debugmode)
    {
         static char debugarg[100];


         (void)sprintf(debugarg, " -x   /tmp/RBRdebug%d    ", getpid(
                                                                    ));
         remdebugflag = debugarg;
    }

    len = strlen(rcmdpath) +
          strlen(bufsizeP) +
          strlen(remdebugflag) +
          strlen(minus_c) +
          strlen(minus_c_arg) +
          strlen(noclobber) +
          1;    /* for '\0' */

    cmd = sm_fmalloc(len);
    (void)sprintf(cmd, "%s%s%s%s%s%s",
                  rcmdpath,
                  bufsizeP,
                  remdebugflag,
                  minus_c,
                  minus_c_arg,
                  noclobber);

    return cmd;
}        /* end of make_remote_cpiogen_cmd() */
```

```
/*
 * 1) Remove rcx references.
 * 2) Research whether ebc_normalize updates.
 */
/*
 * Construct the path name of the remote command that
 * will be executed on the destination client.
 *
 * Return ptr to constructed string.
 * NOTE: caller must copy if string is to be preserved.
 */

static char *
generate_rcmdpath(int SubmitObjectID,
                  int SubmitElemID)
{
    static char *mybuf = NULL;
    size_t len_needed;
    char *pph;   /* pointer to %h */
    char *p;
    char *q;
    char *norm_host;

    char temp_scriptname[SUBMIT_FIELD_MAX];
    char temp_client_hostname[SUBMIT_FIELD_MAX];

    int GetSEStatus = 0;

    if ( GetSERcmdScriptName(SubmitObjectID, SubmitElemID,
                             temp_scriptname, SUBMIT_FIELD_MAX,
                             &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get rcmd script name.");
      return NULL;
    }

    if ( GetSEDestClientName(SubmitObjectID, SubmitElemID,
                             temp_client_hostname, SUBMIT_FIELD_MAX,
                             &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get client destination name.");
      return NULL;
    }
    /*
     * If there is a %h in the rc_client_scriptname,
     * that means put the hostname in there.
     *
     * Sorry, no escapes implemented.  You can't have an
     * rc_client_scriptname with a literal %h in it.  Tough.
     */

    for (pph = temp_scriptname; *pph != '\0'; pph++)
    {
        if (*pph == '%' && *(pph+1) == 'h')
        {
            break;
        }
    }

    /*
     * no %h, just use the bare client scriptname
     */

    if (*pph == '\0')
    {
        return temp_scriptname;
    }

    /*
     * there is a %h ... insert the destination client
     * name into the rc_client_scriptname.  First
     * compute how much storage will be needed to
     * hold the result.
     */

    if (!ebc_can_it_be_normalized(temp_client_hostname))
    {
        rec_api_log_csm(SUB_CSM_NOMEM, NULL);
        rbe_log_stats(
                0, "Could not allocate memory generate_rcmdpath");
        /*      StartDoneCallBack(EP_RB_RECOVER_FATALERR);*/
        return NULL;
    }

    norm_host = ebc_normalize(temp_client_hostname);
    len_needed =
        strlen(temp_scriptname) - 2 +          /* -2: %h */
        strlen(norm_host) + 1;
                                                      /* +1: '\0' */

    if (mybuf == NULL || strlen(mybuf) < (size_t)(len_needed-1))
    {
        if (mybuf != NULL)
        {
            free(mybuf);
        }

        if ((mybuf = malloc((int)len_needed)) == NULL)
        {
            rec_api_log_csm(SUB_CSM_NOMEM, NULL);
             rbe_log_stats(
                    0, "Could not allocate memory generate_rcmdpath");
            return NULL;
        }
    }

    q = mybuf;
    for (q = mybuf, p = temp_scriptname; p < pph; q++, p++)
    {
        *q = *p;
    }

    (void)strcpy(q, norm_host);
    (void)strcat(mybuf, pph+2);

    return mybuf;
}       /* end of generate_rcmdpath() */
```

```
int
StartWorkItemRestore(struct recover_context *rcx,
                     struct auxproc *xp,
                     int SubmitObjectID,
                     int SubmitElemID)
{
    char    *rcmdv[4];    /* 0 hostname, 1 locuser, 2 remuser, 3 cmd */
    int      changed_priv = 0;
    int      xcpiogen_pid;
    rcmd_pkt0_info pkt0p_buffer;
    char *err_str_buffer = NULL;

    char temp_client_hostname[SUBMIT_FIELD_MAX];
    char temp_client_rbuname[SUBMIT_FIELD_MAX];
    long  temp_effective_uid;
    boolean_ty temp_src_sysadmin;

    int GetSEStatus = 0;
    int GetSOStatus = 0;


    if ( GetSEDestClientName(SubmitObjectID, SubmitElemID,
                             temp_client_hostname,
                             SUBMIT_FIELD_MAX,
                             &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get client destination name.");
      return -1;
    }



    if ( GetSEClientUserName(SubmitObjectID, SubmitElemID,
                            temp_client_rbuname,
                            SUBMIT_FIELD_MAX,
                            &GetSEStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get client user name.");
      return -1;
    }



    temp_src_sysadmin = GetSOSourceSystemAdmin(SubmitObjectID,
                                               &GetSOStatus);
    if (0 != GetSOStatus)
    {
      rbe_log_stats(
            0, "Unable to get the source system admin priveledges.");
      return -1;
    }

    if ( GetSOEffectiveUID(SubmitObjectID,
                           &temp_effective_uid,
                           &GetSOStatus) != 0)
    {
      rbe_log_stats(0, "Unable to get effective uid.");
      return -1;
    }


    if (0 != (temp_src_sysadmin) || 0 != temp_effective_uid)
    {
        set_recovery_privileges(rcx, &changed_priv);
    }
    rcmdv[0] = temp_client_hostname;
    rcmdv[1] = temp_client_rbuname;
    rcmdv[2] = temp_client_rbuname;
    rcmdv[3] = make_remote_cpiogen_cmd(rcx,
                                       SubmitObjectID,
                                       SubmitElemID);

    if(NULL == rcmdv[3])
    {
      return -1;
    }
    xcpiogen_pid = start_cpiogen(rcx, xp,
                                 SubmitObjectID,
                                 SubmitElemID,
                                 0,
                                 rcmdv,
                                 &pkt0p_buffer,
                                 &err_str_buffer);

    if (changed_priv)
    {
         reset_recovery_privileges(rcx, changed_priv);
    }
    return xcpiogen_pid;
} /* StartWorkItemRestore() */
```

```
/*
 * Check to see if the user has root access to the
 * destination client. If so, give him/her the root
 * privileges on the recovery.
 */

static void
set_recovery_privileges(struct recover_context *rcx,
                        int *changed)
{
    int root_access;
    eperrno errnum;


    *changed = 0;
    (void)rbc_canirecover(rcx->rc_config, rcx->rc_client_hostname,
                          rcx->rc_human_uidname, &root_access,
                                                              &errnum);
    if (root_access)
    {
         if (debugmode)
         {
             rbe_log_stats(
                    0, "the user is a sys admin for the dest client");
         }

         rcx->rc_recovery_flags |= RC_RECFLAG_DEST_SYSADMIN;
         if (rcx->rc_effective_uid != 0)
         {
             if (debugmode)
             {
                 rbe_log_stats(0, "changing the uid to admin status");
             }

             rcx->rc_effective_uid = 0;
             rcx->rc_effective_uidname = "root";
             *changed = SET_ROOT;
         }
    }
    else
    {
         if (debugmode)
         {
             rbe_log_stats(
            0, "the user don't have sys admin status on dest client");
             rbe_log_stats(
                        0, "effect uid = %d", rcx->rc_effective_uid);
         }

         /*
          * The user does not have root access to the dest
          * client. But s/he is the system admin for the
          * source client, therefore, we need to stripe the
          * root stuff from the user during the recovery.
          */

         rcx->rc_recovery_flags ^= RC_RECFLAG_DEST_SYSADMIN;
         if (rcx->rc_recovery_flags & RC_RECFLAG_SOURCE_SYSADMIN)
         {
             if (debugmode)
             {
                 rbe_log_stats(
                        0, "changing the uid to regular user status");
             }
```

```
                rcx->rc_effective_uid = rcx->rc_human_uid;
                rcx->rc_effective_uidname = rcx->rc_human_uidname;
                *changed = SET_USER;
            }
        }
}       /* end of set_recovery_privileges() */
```

```
/*
 * Reset the user's identity. The user may have root access
 * on the destination client, but not on the source, and vice
 * versa.
 */

static void
reset_recovery_privileges(struct recover_context *rcx,
                          int changed)
{
    if (changed == SET_ROOT)
    {
        if (debugmode)
        {
            rbe_log_stats(
                      0, "resetting the uid to regular user status");
        }

        rcx->rc_effective_uid = rcx->rc_human_uid;
        rcx->rc_effective_uidname = rcx->rc_human_uidname;
    }
    else
    {
        if (debugmode)
        {
            rbe_log_stats(0, "resetting the uid to sys admin status");
        }

        rcx->rc_effective_uid = 0;
        rcx->rc_effective_uidname = "root";
    }
}       /* end of reset_recovery_privileges() */
```

```
/**
 **
 **  FUNCTION DESCRIPTION:
 **
 **      This function will start the workitem restore termination.
 **
 **
 **  INPUTS:
 **      int auxproc_pid -- auxproc's pid.
 **
 **  RETURN VALUE:
 **
 **      none
 **
 **
 **  SIDE EFFECTS:
 **
 **       auxproc sent the USR1 signal, auxproc will send xcpiogen the
 **      TERM signal to quit the restore. Auxproc will wait until the
 **      restore terminates by waiting for the remote command's exit
 **      status.
 **
 **++
 **/

void
QuitWorkItemRestore(int auxproc_pid)
{


    /* Auxproc will now alert xcpiogen by sending the
     * TERM signal.
     * This will give xcpiogen the ability to
     * clean up tmpfiles and clean-up sockets.
     * xcpiogen will commit suicide as a result of
     * this signal.
     */


    /*
     * Alert the auxproc that we want out.
     * recxcpio etc should also die as a result
     * of xx_read_or_die_xx() in recx.  The SIGUSR1
     * should not kill the auxproc itself, but only
     * notify auxproc of the diminishing restore.
     */

    (void)kill(auxproc_pid, SIGUSR1);

    if (debugmode)
    {
        rbe_log_stats(0, "%s %d quiting restore in process",
                        AUXPROCNAME,
                        auxproc_pid);
    }

    /*
     * now that we have indirectly killed the xcpiogen and alerted the
                                                                     a
     * auxproc,
             the signal from the auxproc will be picked up by the next
     * routine, auxprocsig_handler, it will notify the user of the
     * results and does the cleaning up.
```

```
     */
    return;
}       /* QuitWorkItemRestore() */
```

```
/**  GetAuxprocResults()
 **
 **  FUNCTION DESCRIPTION:
 **
 **     Inherited from auxprocsig_handler().
 **
 **      This routine is called when there is information appears
 **     in the aux-process. The aux-process starts xcpiogen and
 **      is responsible for "listening" to status coming from xcpiogen.
 **      When status comes back from xcpiogen, the aux-process signals
 **      this process, which is trapped by the caller. And finally,
 **      this routine is called.
 **
 **  Args:
 **    (I) resfd int -- results file decriptor from auxproc.
 **    (O) results -- work item results.
 **
 **  RETURN VALUE:
 **     The number of local and remote status collected from fd.
 **
 **  SIDE EFFECTS:
 **
 **     none
 **
 **
 **/


int
GetAuxprocResults(int resfd,
                  wi_restore_results *results)
{
    int          exit_stat;
    char         *resbuf;
    int          n_read;
    char         c = 0;
    int         n_status_read = 0;

    while ((n_status_read < 2) && (1 == fd_avail_test(resfd)))
    {
      resbuf = (char *) &exit_stat;

      if(1 != pread_or_warn(resfd, &c, 1, auxproc_comm_warning))
      {
         return 0;
      }

      if (debugmode)
      {
         rbe_log_stats(0, "GetAuxprocResults() called: '%c'", c);
      }

      if (c == 'R')
      {
         if (1 == fd_avail_test(resfd))
         {
           /*
            * the 'R' command is the for the remote process status
            */
           n_read = auxres2(resfd, 'R', sizeof(int), &resbuf);
           if (-1 != n_read )
           {
             results -> remote_exit_status = exit_stat;
             results -> remote_exit_set = TRUE;
             n_status_read++;
```

```
             if (debugmode)
             {
               rbe_log_stats(
                     0, "remote exit status obtained: %d", exit_stat);
             }
           }
           else
           {
             rbe_log_stats(
                 0, "Internal error: remote exit status Incomplete.");
             return -1;
           }
         }
      }
      else if (c == 'r')
      {
         if (1 == fd_avail_test(resfd))
         {
           /*
            * the 'r' command is the for the local process status
            */

           n_read = auxres2(resfd, 'r', sizeof(int), &resbuf);
           if (-1 != n_read )
           {
             results -> local_exit_status = exit_stat;
             results -> local_exit_set = TRUE;
             n_status_read++;
             if (debugmode)
             {
               rbe_log_stats(
                      0, "local exit status obtained: %d", exit_stat);
             }
           }
           else
           {
             rbe_log_stats(
                 0, "Internal error: local exit status Incomplete.");
             return -1;
           }
         }
      }
      /*      sleep (1);*/
    } /* while() */

    return n_status_read;
    /*return n_status_read;*/

}        /* GetAuxprocResults() */
```

```
/*
 * KillWorkItemRestore()
 *
 *  Kill the work item restore. Keep in mind this
 *  may only be done if the work item is not running
 *  a restore.
 *
 *  This routine also does the waitpid for auxproc.
 *  The waitpid (ChildDone()) eliminates the defunct auxproc
 *  process.
 *
 *  If the work item is running then one must do the
 *  following.
 *
 *  1) Call QuitWorkItemRestore()
 *  2) Wait for and read results from the cmd_from pipe.
 *  3) Call KillWorkItemRestore()
 *
 * Args:
 *  ap_pid -- auxproc pid.
 *  cmd_to -- auxproc cmd pipe.
 *
 * Returns:
 *  int -- Zero for success.
 */
int
KillWorkItemRestore(int ap_pid, int cmd_to)
{
  int killRet;
  int apResult;
  char *apName=AUXPROCNAME;
  char *databuf = NULL;
  int ChildDoneRet;

  killRet = kill(ap_pid, SIGTERM);

  if (-1 == killRet)
  {
        rbe_log_stats(
            0, "Can't send sigterm to \"%s\": Pid: %d, error = %s\n",
                        apName, ap_pid, strerror(errno));
        return -1;
  }

  do
  {
     ChildDoneRet = ChildDone(ap_pid, &apResult);
     if(0 == ChildDoneRet)  sleep (1);
  }
  while(0 == ChildDoneRet);

  switch (ChildDoneRet)
  {
    /*  -1 internal error. errno is set.
     *  0  child still running.
     *  1  child exited.
     *  2  child signalled (no core)
     *  3  child signalled core file generated.
     *  4  child stopped.
     */
      case(-1):

        return -1;
        /* no break necessary */
```

```
      case(1):

          rbe_log_stats(0,
                        "Sigterm did not bring down \"%s\": Pid: %d,"
                        "  it instead exitted with = %d\n",
                        apName, ap_pid, apResult);
          break;

      case(2):
      case(3):
        if (SIGTERM != apResult)
         {
          rbe_log_stats(0,
                        "Sigterm did not bring down \"%s\": Pid: %d,,"
                        "  instead killed by signal = %s\n",
                        apName, ap_pid,
                        strsignal(apResult));
        }
        break;

      case(4):
          rbe_log_stats(0,
                        "Sigterm did not bring down \"%s\": Pid: %d,"
                        "  instead stopped by signal = %s\n",
                        apName, ap_pid,
                        strsignal(apResult));

        break;

  }


  /*  auxcmdpacket(cmd_to, 'q', 0, databuf);*/
  return 0;

} /* KillWorkItemRestore() */
```

```
/*
 *              RSLauxmain.c
 *
 * Copyright (c) 1998, 1999 EMC Corporation.  All rights reserved.
 *
 *
 * Table of Contents:
 *--------------------
** static char *rb_getmethod(host, int *concurrency, int *client_type)
** int looprw(int fd, char *buf, int nbytes, int (*func)());
** int fd_avail_1_wait_intr(int fd);
 */

#define  _POSIX_SOURCE 1
#define  E_GRANDFATHER 1
#define TIME_STRUCT
#include <eb/eb_port.h>

#include <netdb.h>
#include <pwd.h>
#include <ctype.h>
#include <sys/wait.h>
#include <util/esl_select.h>
#include <eerrno/esl_strerror.h>
#include <util/esl_limit.h>
#include <stdlib.h>

/* SFP header */
#include <cdl/cdl.h>
#include <cdl/cdl_server.h>

#include <ebconfig/rbconfig.h>
#include <epffdb/ffdb.h>
#include <epffdb/rbfinclient.h>
#include <ebutil/eb_normalize.h>
#include <ebutil/ebsock_if.h>
#include <ebutil/ebutil.h>
#include <eb/rb_log.h>
#include <restore/REprogmsg.h>

#include <RSLinterns.h>
#include <RSLspexits.h>
#include <RSLremfd.h>
#include "RSLauxSupp.h"

/* EDMLINK API */
#include <edmlink/edmlink_api.h>


/* lint ugliness */
#ifndef Printf
#define Printf   (void)printf
#endif
#define MAX_FD 1024
int debugmode = 0;
int is_symmpath = 0;
int sp_cdlexitdone = 0;

static int xcpiogen_pid = -1;
static int xcpiogen_prog_fd = -1;
int logging_channel = -1;

int xcpiogen_pipe[2] = {-1,-1};
static char wifiledir[] = "/usr/epoch/etc/";

static char *rb_getmethod(
```

```
                register char *host, uint_t *concurrency, uint_t *client_type);
static void sigusr1_handler(int sigvalue);
                                                   /* "attention" signal */
static void sigterm_handler(int sigvalue);  /* sigterm signal */
static int write_CDL_no_eintr(int fd, char *buf, int nbytes);
static int read_CDL_no_eintr(int fd, char *buf, int nbytes);

static int ForwardXcpiogenProgress(int xcpiogen_prog_fd,
                                   int restore_engine_prog_fd,
                                   boolean_ty *zero_byte_read);

static int DemuxAuxChildren(int progress_fd,
                            int remote_fd,
                            char *remote_progname,
                            int xcpiogen_fd,
                            int xcpiogen_pid,
                            int *remote_exit);

extern int fd_avail_test(int); /* Found in RSLauxSupp.c */

struct auxproc_context
{
    int        ap_my_auxnum;
    int        ap_r_fd;
    int        ap_w_fd;
    int       ap_r_bulk_fd;
    int       ap_w_prog_fd;
    char      ap_cmd;
    int       ap_datalen;
    char     *ap_data;
    int       ap_resultlen;
    char     *ap_resultdata;
    ushort_t ap_shelltcp_port;
    int       ap_have_shelltcp_port;
    int       ap_socketport;
    char     *ap_sockethost;
    char     *ap_workitem;
    struct rbc_configs *ap_config;
    char      ap_error_message[4096];
};
/*
 * Read remote stderr output from fd.
 * Parse it; act on it according to protocol.
 * Remote exit status is the last thing that comes
 * back via the remote stderr stream; return the remote
 * exit status via *exitp.
 */

enum input_states
{
    INSTATE_NG,
    INSTATE_SEARCH_PREFIX0,
    INSTATE_SEARCH_PREFIXN,
    INSTATE_GATHER_COOKIE,
    INSTATE_SEARCH_SUFFIXN,
    INSTATE_GATHER_STATUS,
    INSTATE_COPY_TO_STDOUT,
    INSTATE_NEWLINE
};
static boolean_ty parse_remote_stderr_info2(int prog_fd,
                                            int remote_fd,
                                            int *exitp,
                                            char *remhostname,
                                            boolean_ty first_call,
                                            enum input_states
                                                     *state_ptr,
                                        enum input_states
                                                *next_state_ptr,
                                        boolean_ty
                                     *skipping_leading_whitespace,
                                        int *parsePos,
                                        int *msgPos);


static char *recover_size_prefix(struct auxproc_context *cxp);
static int  *ebr_direct_rcmd(char **ahost, unsigned short inport,
                             struct auxproc_context *cxp,
                                                char *locuser,
                             char *remuser, char *cmd, int *fd2p);
```

```
/*
 * The auxilliary process(es) communicate with the main
 * process via a simple protocol run on a pair of pipes.
 *
 * The parent writes commands to the auxilliary process.
 * The format of a command is:
 *
 *       <cmd><data-len><data>
 *
 * where
 *       <cmd> is a one-byte command
 *       <data-len> is an "int",
 *                              and indicates the number of <data> bytes
 *       <data> is command-specific data.
 *
 * <data-len> may be zero, but must always be present. Therefore, the
 * minimum command "packet" is five bytes long.
 *
 * Result packets are written back to the parent.
 *                                                    The packet format is
 * the same as the command format, though (obviously) the format
 * of the data in a results packet is usually different from
 * the format in a command packet.
 *
 * Communications are assumed to be error-free.   In other
 * words, pipes are assumed to work correctly.
 */

static int attn_ec;      /* count of SIGUSR1s (ATTN signals) */



/*
 * 'z' prefixes identify top-level routines
 *      called from the auxproc switch.
 *
 *      There is no significant to the 'z' letter -- I just picked
 *      a letter at random (hah!) to identify the top level funcs.
 */

static void z_rcmdfilter(struct auxproc_context *cxp);
static void z_exec_separate_auxproc(struct auxproc_context *cxp,
                                    char *pathname);
```

```
int main(int argc, char *argv[])
{
    char *auxproc_intial_delay_str = NULL;
    int auxproc_intial_delay_int = 0;

    if (argc != 9)
    {
        exit(1);
    }

    auxproc_intial_delay_str = getenv("AUXPROC_INITIAL_DELAY");
    if (NULL != auxproc_intial_delay_str)
    {
      auxproc_intial_delay_int = atoi(auxproc_intial_delay_str);
      sleep(auxproc_intial_delay_int);
    }

    debugmode = atoi(argv[6]);

    do_auxproc(atoi(argv[1]), /* auxproc ordinate. */
               atoi(argv[2]), /* cmd_to auxproc pipe*/
               atoi(argv[3]), /* cmd_from auxproc pipe*/
               atoi(argv[4]), /* bulk_to auxproc pipe*/
               atoi(argv[5]), /* prog_from auxproc pipe*/
               argv[0],       /* progname */
               argv[7],       /* socket host name */
               atoi(argv[8]));/* socket port */

    return(0); /* Can we get here ?? */

} /* End main() */
```

```
/*
 * Worker-bee loop.
 *
 * Read instructions on r_fd.
 * Write results to w_fd.
 */

int ncmds_rcvd = 0;

void
do_auxproc(int procnum,
           int r_fd,
           int w_fd,
           int r_bulk_fd,
           int w_prog_fd,
           char *Xname,
           char *sockethost,
           int socketport)
{
    struct auxproc_context    ctx;
    struct servent           *sp;
    sigset_t                  empty_set;
    int                       rotation_size;
    int                       errnum;
    int                       index_fd;

    /*
     * close all file descriptors that we do not need
     * we only need the ones passed to us from the restore
     * engine.  So we can close anything above stderr and anything
     * that is no equal to what was passed in
     */
    for(index_fd = 3; MAX_FD > index_fd; index_fd++)
      {

        if ((index_fd == r_fd) ||
            (index_fd == w_fd) ||
            (index_fd == r_bulk_fd) ||
            (index_fd == w_prog_fd))
          {
            continue;
          }
        (void)close(index_fd);

      }

    memset(&ctx, 0, sizeof(struct auxproc_context));
    /*
     * Prepare sigaction parameters
     */

    sigemptyset(&empty_set);

    /*
     * ignore mild keyboard interrupts (^C); parent handles
     */

    eb_set_signal_handler(SIGINT, SIG_IGN, &empty_set, E_SA_RESTART);

    /*
     * SIGUSR1 used to get our attention when the parent
     * process really wants us to stop what we are doing.
     */

    eb_set_signal_handler(
                            SIGUSR1, sigusr1_handler, &empty_set, E_SA_RESTART);
    eb_set_signal_handler(
                            SIGTERM, sigterm_handler, &empty_set, E_SA_RESTART);

    /*
     * Arguments passed are collected together in
     * this context structure only because that makes
     * it easier to add/change arguments in the future
     * (diddle the structure rather than diddle a zillion
     * calls in the switch statement).
     */

    ctx.ap_my_auxnum  = procnum;
    ctx.ap_r_fd       = r_fd;
    ctx.ap_w_fd       = w_fd;
    ctx.ap_r_bulk_fd  = r_bulk_fd;
    ctx.ap_w_prog_fd  = w_prog_fd;
    ctx.ap_sockethost = sockethost;
    ctx.ap_socketport = socketport;

    /*
     * Will need the shell/tcp service later; if
     * can't find it now we might as well complain now.
     */

    ctx.ap_have_shelltcp_port = 0;
    if ((sp = getservbyname("shell", "tcp")) != NULL)
    {
        ctx.ap_shelltcp_port = (ushort_t)sp->s_port;
        ctx.ap_have_shelltcp_port = 1;
    }
    else
    {

        WriteFmtStringMsg(
                    ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                        "Error: Cannot find shell/tcp network
                                                        service.\n"
                        "Browsing of catalogs may continue, but"
                        "the \"start\" command will not work.");
    }

    if (NULL == ctx.ap_config)
    {
        if(NULL == (ctx.ap_config = malloc(sizeof(
                                        struct rbc_configs))))
        {
          /* We would prefer to log this stuff, but we
           *  have not opened logging yet.
           */
          WriteFmtStringMsg(
                    ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                            "Could not allocate memory in
                                                    do_auxproc");
          exit(1);
        }
        if (rbc_parse_config(
                    NULL /*use the default name */ , &ctx.ap_config,
                            RBC_PARSE_DO_NOT_PRESERVE |
                                                RBC_PARSE_APPLY) != 0)

        {
          /* We would prefer to log this stuff, but we
           *  have not opened logging yet.
           */
```

```
          rec_api_log_csm(SUB_CSM_NO_PARSE_CFG, NULL);
          WriteFmtStringMsg(
                    ctx.ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                            "Auxproc -- Cannot parse configuration
                                                          file");

          exit(1);
        }
    }

    if ((errnum = eb_path_init()) != 0)
    {
      exit(1);
    }

    (void)rbe_log_init("auxproc");

    /*
     * Determine the "recoveries.log" file rotation size.
     * Use a default unless this was specified in the config file.
     */

    rotation_size = RBRECOVER_LOGSIZE;
    if ((ctx.ap_config != (struct rbc_configs *)NULL)
        && (ctx.ap_config->srlfile.filename != NULL))
    {
        rotation_size = ctx.ap_config->srlfile.rotation_size;
    }

    (void)rbe_log_add(STATS_LOGGING, eb_recover_logpath, LOGT_FILE,
                            rotation_size, &logging_channel);


    /*
     * The actual worker-bee loop
     */

    for (;;)
    {
        char c;
        int datalen;
#define SMALL_DATALEN 128
        char buf[SMALL_DATALEN];
        char *data;

        /*
         * read command byte
         */

        pread_or_die(r_fd, &c, 1, _exit);

        /*
         * read data-len
         */

        pread_or_die(r_fd, (char *)&datalen, sizeof datalen, _exit);

        /*
         * We do the read here for simple commands that do not
         * take much data.
         */

        if (datalen > SMALL_DATALEN || datalen == 0)
        {
            data = NULL;
        }
        else
        {
            pread_or_die(r_fd, buf, datalen, _exit);
            data = buf;
        }

        ++ncmds_rcvd;
        ctx.ap_cmd          = c;
        ctx.ap_datalen      = datalen;
        ctx.ap_data         = data;
        ctx.ap_resultlen    = 0;
        ctx.ap_resultdata   = NULL;

        /*
         * NOTE: This switch statement is in alpha-order
         *       (case-folded) to make it easier for humans
         *       to find entries (at least, it was in alpha-order
         *       when I wrote this comment).
         */

        switch (c)
        {
            /*
             * 'P'
             *  Ping command.  Used for debugging.
             */

        case 'P':
            ctx.ap_resultlen  = ctx.ap_datalen;
            ctx.ap_resultdata = ctx.ap_data;
            break;

            /* 'q'
             *  Quit command.
             */

        case 'q':
            rbe_close_logs(logging_channel);
            _exit(0);

            /*
             * 'r'
             * rcmd filter.  Fire up a process, typically xcpiogen,
             * with output from the process going to an rcmd
             * connection.
             */

        case 'r':
            z_rcmdfilter(&ctx);
            break;

            /* 'X'
             * Exec separate process version of do_auxproc.
             * Used for debugging.
             * No value returned to parent.
             * If exec fails, auxproc dies.
             */

        case 'X':
            z_exec_separate_auxproc(&ctx, Xname);
            _exit(2);
            break;

        default:
            rec_api_log_csm(SUB_CSM_INV_AUXPROC_CMD, NULL);
            rbe_log_stats(
                          0, "auxproc -- invalid command in do_auxproc");
```

```
            exit(1);
            break;
        }   /* end of switch */

        /*
         * Simple commands set ap_resultlen and ap_resultdata
         * and we push the results to the parent here.
         */

        if (ctx.ap_resultlen >= 0)
        {
            pwrite_or_die(w_fd, &c, 1, _exit);
            pwrite_or_die(w_fd, (char *)&ctx.ap_resultlen,
                          sizeof ctx.ap_resultlen, _exit);
            pwrite_or_die(w_fd, ctx.ap_resultdata,
                          ctx.ap_resultlen, _exit);
        }
#if 0
        if('r' == c)
        {
          rbe_log_stats(0, "Auxproc(%d) local exit is %d.", getpid(),
                        ctx.ap_resultlen);

        }
#endif
    }   /* end of for loop */
}       /* end of do_auxproc() */
```

```
/*
 * Fire up a process with stdin from parent and output to rcmd.
 * Protocol traffic between "parent" (main process) and us:
 *
 *      'r' parent --> auxproc     contains local & remote cmd info
 *      '0' auxproc --> parent     returns "setup" result, see below
 *      'R' auxproc --> parent     returns remote exit status (1 int)
 *      'r' auxproc --> parent     returns local exit status  (1 int)
 *
 * The '0' result packet describes the results of setting things up.
 * It contains the following information:
 *
 *      success/failure : int    [ 0 is success,
                                     non-zero is a failure code ]
 *      errnum          : int    [ 0 if no applicable errno code ]
 *      pid             : int    [ process ID of local xcpiogen proc
                                                                  ]
 *      msglen          : int    [ length, in bytes,
                                                  of following str ]
 *      errstr          : string [ string describing failure ]
 *
 * If things were set up successfully, then two integer zero
 * values, a non-zero pid, and one more zero (msglen) are
 * sent in the '0' packet.
 *
 * The defined failure codes are:
 *      TBD
 *
 * The 'R' result packet will not be sent if the '0' result
 * indicates that an error occurred.  The '0' and 'r' result
 * packets are always sent.
 *
 * Parent passes the following in the 'r' packet <data>:
 *       (string) rcmd-hostname
 *       (string) rcmd-locuser
 *       (string) rcmd-remuser
 *       (string) rcmd-cmd
 *       (int)    setuid-val
 *       (int)    filter-cmd-fd-info (explained below)
 *       (int)    future-flags (flags for future hacks;
                                                  always zero now)
 *       (string) filter-cmd
 *       (int)    filter-cmd-argc
 *       (string) filter-cmd-argv0
 *       (string) ...
 *       (string) filter-cmd-argvN
 *      (int)    db API socket info
 *      (string) db API socket host name
 *      (string) filespec from workitem
 *
 * If filter-cmd-fd-info is -1, then stdout on the filter cmd
 * is set up to go to the rcmd.
 * Otherwise, a separate file descriptor (neither stdout nor stderr)
 * is set up to go to the rcmd, and the argv element indicated
 * by filter-cmd-fd-info is used as a sprintf template for passing
 * the file descriptor number to the filter process.
 *
 * For example, if filter-cmd-fd-info is 1, then filter-cmd-argv1
 * should be a sprintf format string, which will be given to
 * sprintf along with one integer to pass the file descriptor number
 * to the filter command.
 *
 * Items which are (string) are '\0' terminated.
 *
 * Parent is responsible for obeying the built-in size limits:
```

```
 *    no more than 100 argv strings for filter-cmd
 *    no more than 10000 total bytes of auxproc data
 *    no more than 100 bytes of filter-cmd-fd-info argv
 */


struct rcmd_pkt0_info
{
    int failcode;
    int errnum;
    int pid;
    int msglen;
    /* variable length char string message follows */
};

/*
 * Function to build command line prefix for adding environment
                                                          variable

 * for log truncation
 */

static char *
recover_size_prefix(struct auxproc_context *cxp)
{
    static char lbuff[128];

    if (cxp->ap_config->crlfile.rotation_size != NO_ROTATION)
    {
        sprintf(lbuff, "EB_MAX_CLIENT_LOG_SIZE=%d;
                                  export EB_MAX_CLIENT_LOG_SIZE;",
                cxp->ap_config->crlfile.rotation_size);
    }
    else
    {
        memset(lbuff, 0, sizeof(lbuff));
    }

    return lbuff;
}       /* recover_size_prefix */
```

```
static void
z_rcmdfilter(struct auxproc_context *cxp)
{
    char        *data;
    char        *rcmd_stuff[4];
                          /* 0 hostname, 1 locuser, 2 remuser, 3 cmd */
    char        *human_name;
    struct rcmd_pkt0_info pkt0;
    char        *pkt0_errstr = "";
    static int   r_resultdata;
    int          filter_cmd_argc;
    int          filter_cmd_fd_info;
    int          filter_cmd_fdout;
    char        *filter_cmd;
    char        *method;
    int           dbseqno=0;
    int           socket_port;
    char          socket_host[100];
    char        *socket_file;
    char          wifilename[200];
    FILE        *infofd;
                                   /* items for socket info file --- */
    static char readmode[] = "r";
    char          cbuf[200];
    char         listener_filename[EB_MAXPATHLEN];
    char        *workitem;
    int          rcmd_stderr = -1;
    int         *rcmd_fd;
    int          rcmd_fds[2];
    int          i,n;
    char         c;
    char        *p2;
    int          wait_result;
    char         fdarg[100];             /* BUILT-IN LIMIT */
    char        *filter_cmd_argv[100];   /* BUILT-IN LIMIT */
    char         databuf[10000];         /* BUILT-IN LIMIT */
    char        *filespec;
    char          sslbuff[200];
                             /* for command with ssl parameter */
    char         holdbuff[1024];
                                  /* hold rbrclient ... command */


    RBC_WORKITEM *pwi;
    RBC_WORKGROUP *pwg;
    boolean_ty xcpiogen_still_running;
    e_errno_ty errnum;



    /* EDMLINK handle */
    ELinkHandlePtr_ty     pELinkHandle     = NULL;

    /* EDMLINK objects */
    ELinkTargetObjPtr_ty pELinkTargetObj = NULL;
    ELinkUserIdObjPtr_ty pELinkUserIdObj = NULL;
    ELinkCmdObjPtr_ty    pELinkCmdObj    = NULL;

    /* EDMLINK global options */
    unsigned long ELinkOptions = 0;

    /* EDMLINK status */
    int ELinkStatus = 0;
    int ELinkStatus_errno = 0;

    data = cxp->ap_data;
```

```
    if (data == NULL)
    {
         /*
          * command dispatcher did not read data for us,
          * there was too much.  We must read it.
          */

         data = databuf;
         pread_or_die(cxp->ap_r_fd, data, cxp->ap_datalen, _exit);
    }

    /*
     * extract the 4 rcmd strings.
     * [3] is the recxcpio cmd.
     */

    for (i = 0; i < 4; i++)
    {
         rcmd_stuff[i] = data;
         data += strlen(rcmd_stuff[i])+1;
    }

    /*
     * extract the remote uidname to be used
     */

    human_name = data;
    data += strlen(human_name) + 1;

    /*
     * extract the filter-cmd-fd-info
     */

    memcpy(&filter_cmd_fd_info, data, sizeof (int));
    data += sizeof (int);

    /*
     * skip the flags
     */

    data += sizeof (int);

    /*
     * extract the filter command
     */

    filter_cmd = data;
    data += strlen(filter_cmd)+1;

    memcpy(&filter_cmd_argc, data, sizeof filter_cmd_argc);
    data += sizeof filter_cmd_argc;

    for (i = 0; i < filter_cmd_argc; i++)
    {
         filter_cmd_argv[i] = data;
         data += strlen(filter_cmd_argv[i])+1;
    }

    filter_cmd_argv[filter_cmd_argc] = NULL;

    /*
     * extract db API socket info
     */

    memcpy((char *)&socket_port, data, sizeof (int));
    data += sizeof (int);

    /*
     * extract db API socket host name
     */

    socket_file = data;
    data += strlen(socket_file) + 1;

    /*
     * extract filespec
     */

    filespec = data;
    data += strlen(filespec) + 1;

    /*
     * extract workitem
     */

    workitem = data;
    cxp->ap_workitem = workitem;
    data += strlen(workitem) + 1;

    pkt0.failcode = 0;
    pkt0.errnum   = 0;
    pkt0.pid      = -1;
    pkt0.msglen   = -1;
    pkt0_errstr   = "";

    /*
     * locate work item in config info
     */

    if (NULL == cxp->ap_config)
    {
        if(NULL == (cxp->ap_config = malloc(sizeof(
                                        struct rbc_configs))))
        {
            rbe_log_stats(
                    0, "Could not allocate memory in z_rcmdfilter!");
            exit(1);
        }
        if (rbc_parse_config(
                    NULL /*use the default name */ , &cxp->ap_config,
                        RBC_PARSE_DO_NOT_PRESERVE |
                                        RBC_PARSE_APPLY) != 0)

        {
            rec_api_log_csm(SUB_CSM_NO_PARSE_CFG, NULL);
            rbe_log_stats(
                    0, "auxproc -- Cannot parse configuration file");
            exit(1);
        }
    }

    for (pwg = cxp->ap_config->pgrouplist; ; pwg = pwg->next)
    {
        if (pwg == (RBC_WORKGROUP *)NULL)
        {
            char csm_err_msg[256];

            rbe_log_stats(
                0, "\n ** No work item name \"%s\" in configuration file",
                            cxp->ap_workitem);
            sprintf(csm_err_msg,
```

```
                    "Cannot restore. Work item not in eb.cfg (
                                        No work item for %s.)",
                    cxp->ap_workitem);
            rec_api_log_csm(SUB_CSM_NO_WITEM_CFG, csm_err_msg);
            rbe_log_stats(0, "Cannot continue without a work item");
            exit(1);
        }

        for (pwi = pwg->pwilist; NULL != pwi; pwi = pwi->next)
        {
            if (0 == strcmp(pwi->name, cxp->ap_workitem))
            {
                goto gotit;
            }
        }
    }

gotit:
    /*
     * connect by direct or rsh methods
     */

    if ((method = rb_getmethod(rcmd_stuff[0], NULL, NULL)) == NULL)
    {
      WriteFmtStringMsg(
                  cxp ->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_WARNING, 0,
                        "Unable to get connection method to host
                                                       \"%s\": "

                        "defaulting to \"rsh\" method\n",
                        rcmd_stuff[0]);
      method = "rsh";
    }

    /*
     * now check for a work item override of the connection method
     */

    if (CNCTN_RSH != pwi->connection_type) /* if other than rsh */
    {
      switch (pwi->connection_type)
       {
       case CNCTN_RSH:
           method = "rsh";
           break;

       case CNCTN_EDMLINK:
           method = "edmlink";
           break;

       case CNCTN_DIRECT:
           method = "direct";
           break;

       case CNCTN_SOCKET:
           method = "socket";
           break;

       case CNCTN_NETWARE:
           method = "netware";
           break;

       default:
           method = "???";
           break;
       }
```

```
        if (debugmode)
          {
              rbe_log_stats(
0, "using connection method \"%s\" due to work item \"%s\" override",
                       method, pwi->name);
          }
    }

    if (NULL != pwi && pwi->recovery_init) /* if a recovery init
                                              command specified */
    {
        (void)system(pwi->recovery_init);
    }

    /* OSGsw34952 -- Workaround for STPclose() behavioral defect
       If pwi->ssl_groupname is not NULL we are going to be using
       SP for the restore.
       Shut off the SSL (STP) exit handler.

       Environment variable is used here so the call to
       CDL_noatexit can be avoided, if desired.
    */

    if ( NULL == getenv("AUXPROC_DONOT_PERFORM_CDLNOATEXIT") )
    {
      if ( NULL != pwi->ssl_groupname )
      {
#if 0
        if ( NULL != getenv("EDM_STP_LOGGING"))
        {
          (void) putenv("STP_EVTLEVEL=DEBUG2");
          (void) putenv("STP_EVTLOG_SHARE=OFF");
        }
#endif

        (void) CDL_noatexit();
        is_symmpath = 1;
      }
    }

    if (0 == strcmp(method, "direct"))
    {
        /*
         * direct connection method here
         */

        if (debugmode)
          {
              rbe_log_stats(0,"Invoking cmd with %s, %s, %s, %s\n",
                            rcmd_stuff[0], rcmd_stuff[1],
                            rcmd_stuff[2], rcmd_stuff[3]);
          }

        strcpy(cbuf,  recover_size_prefix(cxp));

        if (strlen(cbuf) > (size_t)0)
          {
              sprintf(holdbuff, "( %s %s )", cbuf, rcmd_stuff[3]);
              rcmd_stuff[3] = holdbuff;
          }

        rcmd_fd = ebr_direct_rcmd(
                        &rcmd_stuff[0], cxp->ap_shelltcp_port, cxp,
                                rcmd_stuff[1], rcmd_stuff[2],
                                                    rcmd_stuff[3],
```

```
                                            &rcmd_stderr);
        if (debugmode)
          {
              rbe_log_stats(
                    0, "AUXPROC: rcmd returned fd %d, stderr fd %d\n",
                         ((
                    NULL != rcmd_fd) ? rcmd_fd[0] : -1), rcmd_stderr);
          }

        if (rcmd_fd == NULL)
          {
              pkt0.failcode = 2;
              pkt0_errstr = "cannot set up remote connection";

              goto send_0;
          }
    }
    else if ( (0 == strcmp(method, "rsh")) || (0 == strcmp(
                                               method, "edmlink")) )
    {
        /* variable for dealing with symm path and cross restore */

        char sslName[CDL_HOST_LENGTH];
        char sslGroup[CDL_STG_LENGTH];
        char **stgNames;     /* for returned array of STG names */
        int  numGroups = 0;
                              /* for returned number of STG names */
        int  addIndex = 0;
        int  rc = 0;
        boolean_ty  symmPathOK = FALSE;
        /*
         * rsh connection method here
         */

        /*
         * Set up the rcmd connection (rsh method) to the client.
         */

        if (! cxp->ap_have_shelltcp_port)
          {
              struct servent *sp;

              /*
               * Try to get the port number again.
               */

              if ((sp = getservbyname("shell", "tcp")) != NULL)
              {
                  cxp->ap_shelltcp_port = (ushort_t)sp->s_port;
                  cxp->ap_have_shelltcp_port = 1;
              }
              else
              {
                  pkt0.failcode = 1;
                  pkt0_errstr = "shell/tcp service not found";
                  goto send_0;
              }
          }

          /*
           * check for SSL enabled work item and if present add
           * SSL information to command string as an environment
           * variable--but first check for cross restore and make
           * appropriate adjustments
           */
        if (pwi->ssl_groupname != NULL)
        {
            if (!ebc_same_host(
                    rcmd_stuff[0], pwi->sysname)) /* if x-recovery */
            {
                /* Grab the STG entries for this client */
                rc = CDL_getavailablegroups(
                            rcmd_stuff[0], &numGroups, &stgNames);
                if ((numGroups==0) || (rc < 0))
                {
                  char errStr[128] ;
                  sprintf(
    errStr, "Unable to use SymmPath for cross restore to host %s--using
                                  network", rcmd_stuff[0]);
                  rbe_user_error(0, errStr);
                  /*
                   * Default to network by leaving SP Flag set to
                                                               False.
                   */
                }
                else
                {
                    symmPathOK = TRUE;
                   /*
                   * The destination client is SP enabled.
                                                         First check
                   * to see if the same STG group exists.
                                                          If we don't
                   * find the same one we will use the first,
                                                              so set
                   * that as a default
                   */

                    memset(sslGroup,0,CDL_STG_LENGTH);
                    strncpy(sslGroup,stgNames[0],CDL_STG_LENGTH-1);

                   /* Initialize the index */
                   addIndex = numGroups;
                   while (--addIndex >= 0)
                      if (0 == strcmp(
                          pwi->ssl_groupname, stgNames[addIndex]))
                      {
                        strncpy(
                    sslGroup,stgNames[addIndex],CDL_STG_LENGTH-1);
                        continue;
                      }


                   sprintf(sslbuff, "EB_SSL_RECOVER=\"%s\" ;
                                export EB_SSL_RECOVER ;  ",
                        sslGroup);


                }
            }
            else  /* not a cross recovery--just normal SP */
            {
                symmPathOK = TRUE;
                memset(sslGroup,0,CDL_STG_LENGTH);
                strncpy(
                    sslGroup,pwi->ssl_groupname,CDL_STG_LENGTH-1);
                sprintf(sslbuff, "EB_SSL_RECOVER=\"%s\" ;
                                 export EB_SSL_RECOVER ;  ",
                    sslGroup);

            }
```

```
        }



        if (debugmode)
        {
            sprintf(cxp->ap_error_message,
                    "Calling ELinkShell with %s, %d, %s, %s, %s",
                    rcmd_stuff[0], cxp->ap_shelltcp_port,
                                                     rcmd_stuff[1],
                    rcmd_stuff[2], rcmd_stuff[3]);

            rbe_log_stats(0, "%s", cxp->ap_error_message);
            cxp->ap_error_message[0] = 0;
        }

        strcpy(cbuf,  recover_size_prefix(cxp));

        if (strlen(cbuf) > (size_t)0)
        {
            if ( symmPathOK == TRUE)
            {
                sprintf(holdbuff, "(
                          %s %s %s)", cbuf, sslbuff, rcmd_stuff[3]);
                rcmd_stuff[3] = holdbuff;
            }
            else
            {
                sprintf(holdbuff, "( %s %s )", cbuf, rcmd_stuff[3]);
                rcmd_stuff[3] = holdbuff;
            }

        }
        else if (symmPathOK == TRUE)
        {
            sprintf(holdbuff, "( %s %s)", sslbuff, rcmd_stuff[3]);
            rcmd_stuff[3] = holdbuff;
        }

        /* if not Symm Path and cbuf is NULL rcmd_stuff[3] stays
                                                          unchanged */



        /*
        ** EDMLINK API
        */

        /* set the initial edmlink transport methods */
        ELinkOptions = ELINK_SHELL_RCMD | ELINK_SHELL_REXEC;

        /* enable the edmlink transport method */
        if( 0 == strcmp(method, "edmlink") )
        {
            ELinkOptions |= ELINK_SHELL_EDMLINK;
        }

        if( debugmode )
        {
            /* turn on edmlink debugging */
            ELinkOptions |= ELINK_LOGLVL_DEBUG;
        }

        /* initialize the edmlink api */
        pELinkHandle = ELinkInitAPI( ELinkOptions );
        if( NULL == pELinkHandle )
        {
            sprintf(cxp->ap_error_message,
                    "ERROR: Could not initialize the EDMLINK API.");

            WriteFmtStringMsg(cxp-> ap_w_prog_fd,
                              EDMREPROGMSG_AUXPROC_ERROR, 0,
                              "%s\n", cxp->ap_error_message);

            rbe_log_stats(0, "%s", cxp->ap_error_message);
            return;
        }

        /* get new target host object */
        pELinkTargetObj = ELinkNewTargetObj( pELinkHandle,
                rcmd_stuff[0] );

        if( NULL == pELinkTargetObj )
        {
            sprintf(cxp->ap_error_message,
                    "ERROR: Could not create a new EDMLINK target
                                                        object.");


            WriteFmtStringMsg(cxp-> ap_w_prog_fd,
                              EDMREPROGMSG_AUXPROC_ERROR, 0,
                              "%s\n", cxp->ap_error_message);

            rbe_log_stats(0, "%s", cxp->ap_error_message);
            /* clean up and return */
            (void) ELinkDoneAPI( pELinkHandle );
            return;
        }

        /* get new user id object */
        pELinkUserIdObj = ELinkNewEbrUserIdObj( pELinkHandle,
                                                pELinkTargetObj,
                                                rcmd_stuff[1] );

        if( NULL == pELinkUserIdObj )
        {
            WriteFmtStringMsg(cxp-> ap_w_prog_fd,
                              EDMREPROGMSG_AUXPROC_ERROR, 0,
                              "Could not create a new EDMLINK user id
                                                      object.\n" );
            /* clean up and return */
            if( NULL != pELinkTargetObj )
                (void) ELinkDestroyObj(
                                pELinkHandle, pELinkTargetObj );
            (void) ELinkDoneAPI( pELinkHandle );
            return;
        }

        /* get new command object */
        pELinkCmdObj = ELinkNewCmdObj( pELinkHandle,
                pELinkTargetObj,
                rcmd_stuff[3] );

        if( NULL == pELinkCmdObj )
        {
            sprintf(cxp->ap_error_message,
                    "ERROR: Could not create a new EDMLINK command
                                                        object.");


            WriteFmtStringMsg(cxp-> ap_w_prog_fd,
                              EDMREPROGMSG_AUXPROC_ERROR, 0,
                              "%s\n", cxp->ap_error_message);
```

```
            rbe_log_stats(0, "%s", cxp->ap_error_message);
            /* clean up and return */
            if( NULL != pELinkTargetObj )
                (void) ELinkDestroyObj(
                                pELinkHandle, pELinkTargetObj );
            if( NULL != pELinkUserIdObj )
                (void) ELinkDestroyObj(
                                pELinkHandle, pELinkUserIdObj );
            (void) ELinkDoneAPI( pELinkHandle );
            return;
        }

        rcmd_fd = rcmd_fds; /* set valid pointer */

        ELinkStatus = ELinkShell( pELinkHandle,
                pELinkTargetObj,
                pELinkUserIdObj,
                pELinkCmdObj,
                &rcmd_fd[0],
                &rcmd_stderr );
        ELinkStatus_errno = errno;

        rcmd_fd[1] = rcmd_fd[0]; /* this one is bi-directional */

        if (debugmode)
        {
            rbe_log_stats(
                    0, "AUXPROC: ELinkShell returned %d, errno %d,"
                       " fd %d, stderr fd %d\n", ELinkStatus,
                       ELinkStatus_errno, rcmd_fd[0], rcmd_stderr);
        }

        /*
        ** EDMLINK: clean up and shut down the edmlink api
        */

        if( NULL != pELinkTargetObj )
            (void) ELinkDestroyObj( pELinkHandle, pELinkTargetObj );

        if( NULL != pELinkUserIdObj )
            (void) ELinkDestroyObj( pELinkHandle, pELinkUserIdObj );

        if( NULL != pELinkCmdObj )
            (void) ELinkDestroyObj( pELinkHandle, pELinkCmdObj );

        (void) ELinkDoneAPI( pELinkHandle );

        if (0 != ELinkStatus)
        {
            sprintf(cxp->ap_error_message,
                    "cannot set up remote connection when calling "
                    "ELinkShell with %s, %d, %s, %s, \"%s\",
                                                error \"%s\" (%d)",
                    rcmd_stuff[0], cxp->ap_shelltcp_port,
                                                    rcmd_stuff[1],
                    rcmd_stuff[2], rcmd_stuff[3],
                    esl_strerror(
                            ELinkStatus_errno), ELinkStatus_errno);

            pkt0.failcode = 2;
            pkt0_errstr = cxp->ap_error_message;

            if (debugmode)
            {
                rbe_log_stats(
                              ELinkStatus_errno, "ELinkShell failed!\n");
            }

            goto send_0;
        }

        /* add remainder of Symmetrix Path handshaking here
          if the we made it through the other Symm Path tests
        */
        if (symmPathOK == TRUE)
        {
          int newfd;  /* fd for SSL sopen call */

          /*
           * We're going to grab the short name SSL alias for the
           * long network name.
                                 Useless unless this is a cross-restore.
           * (
             because rcmd_stuff[0] == pwi->ssl_groupname on a regular
           *  restore)
           */
          memset(sslName, 0, CDL_HOST_LENGTH);
          strncpy(sslName, rcmd_stuff[0], CDL_HOST_LENGTH-1);

          if ((strlen(rcmd_stuff[0]) >= CDL_HOST_LENGTH) &&
              (0 >= CDL_getsslhostname(sslName, rcmd_stuff[0])))
            {
              char errStr[128] ;
              sprintf(
      errStr, "Unable to determine the short SymmPath hostname of %s",
                                                         rcmd_stuff[0]);
              rbe_user_error(0, errStr);
              pkt0.failcode = 2;
              pkt0_errstr = "Cross-restore is configured to go
                              through SP, yet destination host is not"
                " properly configured in edm.conf";

              goto send_0;
            }

          /*
          * Establish the SymmPath channel
          */
          if ((newfd = CDL_client(rcmd_fd[0],
                                  sslName,
                                  sslGroup)) <0)

          {
            rbe_user_error(0,
                           "Unable to open an SSL listener connection
                                          -- CDL_client failed");
            pkt0.failcode = 2;
            pkt0_errstr = CDL_errstr(newfd);


            goto send_0;
          }

          /* We connected...Use it.. */
          rcmd_fd[0] = rcmd_fd[1] = newfd ;

        }


    }
    else if (0 == strcmp(method, "netware"))
```

```
    {
        /*
         * netware connection method here
         */

        char buf9[EB_MAXPATHLEN];
        char targetbuf[EB_MAXPATHLEN];
        char *tsa = "?";
        char *target = NULL;


        filter_cmd_argv[filter_cmd_argc++] = "-/";
                                                  /* always full path */
        filter_cmd_argv[filter_cmd_argc] = NULL;

       /*
        * Set up the rcmd connection (rsh method) to the client.
        */

       if (! cxp->ap_have_shelltcp_port)
        {
          struct servent *sp;

          /*
           * Try to get the port number again.
           */

          if ((sp = getservbyname("shell", "tcp")) != NULL)
           {
             cxp->ap_shelltcp_port = (ushort_t)sp->s_port;
             cxp->ap_have_shelltcp_port = 1;
           }
          else
           {
             pkt0.failcode = 1;
             pkt0_errstr = "shell/tcp service not found";
             goto send_0;
           }
        }

       rcmd_fd = rcmd_fds; /* set valid pointer */

        /*
         * now skip over "./host/bin/startrec"
         */

        for (p2 = rcmd_stuff[3]; '\0' != *p2 && ' ' != *p2; p2++)
        {
            /* skip ./host/bin/startrec */
        }

        while (' ' == *p2)
        {
            p2++; /* advance to start of next word */
        }

        /*
         * if a -c target:/path then take out target:
         */

        *targetbuf = '\0';
        if (strlen(p2) >=2)
          {
            if (('-' == *p2) && ('c' == p2[1]))
              {
                char *p3 = p2+2;
```

```
                    char *p4;
                    while (' ' == *p3)
                      {
                        p3++; /* advance to start of next word */
                      }
                    p4 = p3; /* save this as start of the prefix */
                    for (;'\0' != *p3; p3++)
                      {
                        if ((':' == *p3) && ('/' == p3[1])) ||
                             ((':' == *p3) && ('\"' == p3[1])))
                          {
                            break;
                          }
                      }
                    if (*p3 != 0) /* if found the prefix */
                      {
                        char * tempChar = NULL;
                        strncpy(targetbuf, p4, p3-p4);
                        targetbuf[p3-p4] = '\0';
                        target = targetbuf;
                        if('\"' == p3[1])
                          {
                            p3++;
                          }
                        tempChar = strchr(p3+1,'\"');
                        if (NULL != tempChar)
                          {
                            tempChar[0] = ' ';
                          }
                        strcpy(p4, p3+1);
                      }
                  }
              }

          /*
           * now, if it's a netware cross recovery, get password,
                                                            etc. from
           * destination target -- don't use source target from pw.
           * pwi points to work item config struct backup was done
                                                                under
           */
          if (0 != strcmp(
                        rcmd_stuff[0], pwi->sysname)) /* if x-recovery */
          {
              RBC_WORKITEM *pwi2 = pwi;


              tsa = "?";
              if (*targetbuf != 0)
              {
                  target = targetbuf;
              }

              for (pwg = cxp->ap_config->pgrouplist; NULL != pwg;
                                                      pwg = pwg->next)
              {
                  for (pwi = pwg->pwilist; NULL != pwi; pwi = pwi->next)
                    {
                        if (ebc_same_host(pwi->sysname, rcmd_stuff[0]))
                        {
                            goto gotit2;
                        }
                    }
              }
gotit2:
```

```
              /*
               * if we did not find a wi for the target system,
               * put user prompting here.
               */

              if (NULL == pwi)
              {
                  pwi = pwi2; /* restore old value if ! found */
              }
          }

          /*
          * check for an encrypted password
           */
          if (NULL != pwi && (
                            pwi->flags & WORKITEM_FLAGS_ENCRYPT_PASSWD))
          {
              sprintf(
           buf9, "r %s -target %s -tsa %s -login %s -epassword %s", p2,
                       ((target != NULL)
                        ? target
                        : ((
            pwi->nw_clnt_target != NULL) ? pwi->nw_clnt_target : "?")),
                       tsa,
                       ((
                  pwi->nw_username != NULL) ? pwi->nw_username : "?"),
                       ((
                       pwi->nw_passwd != NULL) ? pwi->nw_passwd : "?"));
          }
          else
          {
              sprintf(
            buf9, "r %s -target %s -tsa %s -login %s -password %s", p2,
                       ((target != NULL)
                        ? target
                        : ((
            pwi->nw_clnt_target != NULL) ? pwi->nw_clnt_target : "?")),
                       tsa,
                       ((
                  pwi->nw_username != NULL) ? pwi->nw_username : "?"),
                       ((
                       pwi->nw_passwd != NULL) ? pwi->nw_passwd : "?"));
          }

          if (NULL != pwi && (CNCTN_RSH != pwi->connection_type))
          {
              if (CNCTN_NETWARE != pwi->connection_type)
              {
                 WriteFmtStringMsg(cxp->ap_w_prog_fd,
                                   EDMREPROGMSG_AUXPROC_WARNING, 0,
                                   "work item \"%s\" specifies
                                        connection method %s but client "
                                   "was installed to use the netware
                                        method -- using netware method",
                                   pwi->name,
                                   ((
                            CNCTN_RSH == pwi->connection_type) ? "Rsh" :
                                    ((
                    CNCTN_EDMLINK == pwi->connection_type) ? "Edmlink" :
                                     ((
                      CNCTN_DIRECT == pwi->connection_type) ? "Direct" :
                                      ((
                      CNCTN_SOCKET == pwi->connection_type) ? "Socket" :
                                       ((
                    CNCTN_NETWARE == pwi->connection_type) ? "Netware" :
                                        "???"))))));
            }

            if (0 != pwi->connection_port) /* if port specified */
              {
                if (debugmode)
                {
                   rbe_log_stats(
                              0, "using socket port %d to connect to"
                                 " host \"%s\", witem \"%s\"",
                                 pwi->connection_port, pwi->sysname,
                                                        pwi->name);
                }
               cxp->ap_shelltcp_port = (ushort_t)pwi->connection_port;
              }
         }

        if (debugmode)
         {
            rbe_log_stats(
                       0, "Calling nwrcmd with %s, %d, %s, %s, %s\n",
                            rcmd_stuff[0], cxp->ap_shelltcp_port,
                            rcmd_stuff[1], rcmd_stuff[2], buf9);
         }

        rcmd_fd[0] = nwrcmd(&rcmd_stuff[0], cxp->ap_shelltcp_port,
                              rcmd_stuff[1], rcmd_stuff[2], buf9,
                                                       &rcmd_stderr,
                            pwi->ssl_groupname, pwi->ssl_clientname,
                                                               TRUE);
        rcmd_fd[1] = rcmd_fd[0]; /* this one is bi-directional */

        if (debugmode)
         {
            rbe_log_stats(
                   0, "AUXPROC: nwrcmd returned fd %d, stderr fd %d\n",
                            rcmd_fd[0], rcmd_stderr);
         }

        if (rcmd_fd[0] == -1)
         {
            pkt0.failcode = 2;
            pkt0_errstr = "cannot set up remote connection";

             goto send_0;
         }
    }
    else if (0 == strcmp(method, "socket"))
    {
        /*
         * socket connection method here
         */

        if (CNCTN_RSH != pwi->connection_type) /* if other than rsh */
         {
            if (CNCTN_SOCKET != pwi->connection_type)
             {
               WriteFmtStringMsg(cxp->ap_w_prog_fd,
                                 EDMREPROGMSG_AUXPROC_WARNING, 0,
                                 "work item \"%s\" specifies
                                          connection method %s but "
                                 "client was installed to use the
                                socket method -- using socket method",
                                pwi->name,
                                ((
                      CNCTN_RSH == pwi->connection_type) ? "Rsh" :
                                  ((
```

```
                   CNCTN_EDMLINK == pwi->connection_type) ? "Edmlink" :
                                    ((
                     CNCTN_DIRECT == pwi->connection_type) ? "Direct" :
                                    ((
                     CNCTN_SOCKET == pwi->connection_type) ? "Socket" :
                                    ((
                  CNCTN_NETWARE == pwi->connection_type) ? "Netware" :
                                    "???"))))));
             }

            /*
             * read file with name of WI (
                         written by listener) to get info about client
             */

            socket_port  = -1;
            dbseqno = 1;
            strcpy(wifilename, wifiledir);
            if (NULL == socket_file)
              {
                rbe_log_stats(
                         0, "No socket file name passed to auxproc\n");
                return;
              }
            strcat(wifilename, socket_file);

            /*
             * Get socket information from eblistend info file
             */
            errnum = eb_parse_listener_info_file (wifilename,
                                                   cbuf,
                                                   socket_host,
                                                   &socket_port,
                                                   &dbseqno,
                                                   listener_filename);

            if (errnum != E_SUCCESS)
            {
                /*
                 * The routine already logged an error message,
                                                                simply
                 * return the error to caller
                 */
                rbe_log_stats(0,
                              "Unable to read eblistend info file
                                                             \"%s\"\n",
                              wifilename);
                return;
            }

            if (socket_port == -1)
           {
                rbe_log_stats(
                          0, "Invalid port field seen for client\n");
                return;
           }

            if (
      0 != pwi->connection_port) /* if port specified in work item */
           {
                if (debugmode)
                {
                    rbe_log_stats(
                              0, "using socket port %d to connect to"
                                 " host \"%s\", witem \"%s\"",
                               pwi->connection_port, pwi->sysname,
                                                   pwi->name);
                }
                socket_port = pwi->connection_port;
            }
        }

        if (debugmode)
        {
            rbe_log_stats(
  0, "host= %s port=%d seq#=%d\n", socket_host, socket_port, dbseqno);
        }

         rcmd_fd = rcmd_fds; /* set valid pointer */

         /* We need to fix this up...
                                    Modify sopen to call the right type
                 of socket (STP) */

        rcmd_fd[0] = sopen(socket_host, 'c', socket_port);
        if (rcmd_fd[0] < 0)
        {
            rbe_log_stats(0, "Socket Open error: %d\n", rcmd_fd[0]);
            _exit(2);
        }

        /*
         * set up ONLY remote command file descriptors to talk to
                                                              client
         * via socket connection. DO NOT overwrite filter_cmd fd's
         */

        rcmd_fd[1]  = dup(rcmd_fd[0]);

         /* If we are connected,
                          determine if the user wants normal sockets
          * or ssl.
          */

         if (rcmd_fd[0] != -1)
         {
             if (!pwi->ssl_groupname)
             {
                 /*
                  * send OK status to client waiting on data socket
                                                              before
                  * data stream
                  */

                 sprintf(cbuf,"OK: %d socket connection made;
                                                        ", dbseqno);
                 (void)swrite (rcmd_fd[0], cbuf, (int)strlen(cbuf));
             }
             else /* ssl connect */
             {
                 char sslName[CDL_HOST_LENGTH];
                 char sslGroup[CDL_STG_LENGTH];
                 char **stgNames;
                                 /* for returned array of STG names */
                 int  numGroups = 0;
                               /* for returned number of STG names */
                 int  addIndex = 0;
                 int  rc = 0;

                 /* First test to see if this is a cross restore. If
               * so we can't just use the STG group from the source
                                                              client
```

```
                  * since it may not exist on the destination client.
                  * We need to see if the destination is SP enabled and
                  * get a good STG group for that.
                                                We'll use the same group
                  * if it is available.
                                      If no STG group is available we will
                  * fall back to network.
                  */

                  if (!ebc_same_host(
                      socket_host, pwi->sysname)) /* if x-recovery */
                  {
                      /* Grab the STG entries for this client */
                      rc = CDL_getavailablegroups(
                                socket_host, &numGroups, &stgNames);
                      if ((numGroups==0) || (rc < 0))
                      {
                        char errStr[128] ;
                        sprintf(
errStr, "Unable to use SymmPath for cross restore to host %s--using
                                          network", socket_host);
                        rbe_user_error(0, errStr);
                        /*
                         * Default to network and send normal
                                                          handshake.
                         * send OK status to client waiting on data
                                                       socket before
                         * data stream
                         */

                        sprintf(cbuf,"OK: %d socket connection made;
                                                        ", dbseqno);
                        (void)swrite (rcmd_fd[0], cbuf, (int)strlen(
                                                              cbuf));

                      }
                      else
                      {
                          /*
                          * The destination client is SP enabled.
                                                          First check
                          * to see if the same STG group exists.
                                                          If we don't
                          * find the same one we will use the first,
                                                              so set
                          * that as a default
                          */

                           memset(sslGroup,0,CDL_STG_LENGTH);
                           strncpy(
                              sslGroup,stgNames[0],CDL_STG_LENGTH-1);

                          /* Initialize the index */
                          addIndex = numGroups;
                          while (--addIndex >= 0)
                             if (0 == strcmp(
                             pwi->ssl_groupname, stgNames[addIndex]))
                             {
                                 strncpy(
                    sslGroup,stgNames[addIndex],CDL_STG_LENGTH-1);
                                 continue;
                             }

                           /*
                          * We need to get a short name SSL alias if
                                                      the destination
                              * name happens to be long
                              */

                             memset(sslName, 0, CDL_HOST_LENGTH);
                             strncpy(
                               sslName, socket_host, CDL_HOST_LENGTH-1);

                             if ((strlen(
                                    socket_host) >= CDL_HOST_LENGTH) &&
                                 (0 >= CDL_getsslhostname(
                                                 sslName, socket_host)))

                             {
                               char errStr[128] ;
                               sprintf(
          errStr, "Unable to determine the short SymmPath hostname of %s",
                                                         socket_host);
                               rbe_user_error(ffdb_errnum, errStr);
                               pkt0.failcode = 2;
                               pkt0_errstr = "Cross-restore is
                configured to go through SP, yet destination host is not"
                               " properly configured in edm.conf";

                               goto send_0;
                             }
                             if (ebsock_server_connect_ssl(sslGroup,
                                                           sslName,
                                                           &rcmd_fd[0],
                                                            &rcmd_fd[1],
                                                           socket_host,
                                                           dbseqno,
                                                           holdbuff) <
                                                                       0)
                               {
                                 WriteFmtStringMsg(cxp->ap_w_prog_fd,

                                          EDMREPROGMSG_AUXPROC_ERROR, 0,
                                              /* "%s",*/ holdbuff);
                                 return;
                               }
                           }
                     }
                     else
                       {
                         /*
                          * Just a normal Sp restore--nothing special to
                                                                        do
                          */

                         if (ebsock_server_connect_ssl(
                                                  pwi->ssl_groupname,
                                                   pwi->ssl_clientname,
                                                  &rcmd_fd[0],
                                                         &rcmd_fd[1],
                                                  socket_host,
                                                  dbseqno,
                                                  holdbuff) < 0)
                           {
                             WriteFmtStringMsg(cxp->ap_w_prog_fd,
                                          EDMREPROGMSG_AUXPROC_ERROR,
                                                                    0,
                                       /* "%s",*/ holdbuff);
                             return;
                           }
                       }
                 }
```

```
          }

          /* Now that we are finally connected, dup to std err */
        rcmd_stderr = CDL_dup(rcmd_fd[0]);

    }
    else
    {
        /*
         * unknown connection method here
         */
      WriteFmtStringMsg(
                      cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                          "Bad connection method \"%s\"\n", method);
      return;
    }

    if( 0 != pipe(xcpiogen_pipe))
    {
      WriteFmtStringMsg(
                      cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                          "Could not set up progress channel from
                                                          xcpiogen.\n");
      _exit(2);
    }

    switch (xcpiogen_pid = pkt0.pid = fork())
    {
    case -1:
        /* error */
       (void) close(xcpiogen_pipe[0]);
       (void) close(xcpiogen_pipe[1]);
        pkt0_errstr = "fork failed";
        pkt0.failcode = 3;
        pkt0.errnum = errno;
        break;

    case 0:
        /* child */

        /*
         * Make our stdin be the bulk-from-parent stream.
         */
        (void)dup2(cxp->ap_r_bulk_fd, 0);

        if (debugmode)
        {
            rbe_log_stats(
                     0, "FILTER CMD: \"%s\", human_name: \"%s\"\n",
                         filter_cmd, human_name);
            for (i = 0; i < filter_cmd_argc; i++)
            {
                rbe_log_stats(0, "\t%s\n", filter_cmd_argv[i]);
            }
        }

        /*
         * If no fd info passed to child,
                                       then child stdout goes to the
         * rcmd process.  Otherwise, arbitrary fd is used,
                                                        and we pass
         * the fd number as an argument per the format given to us
         */

        if (filter_cmd_fd_info == -1)
        {
                filter_cmd_fdout = 1;
                (void)dup2(rcmd_fd[0], 1);
            }
            else
            {
                filter_cmd_fdout = rcmd_fd[0];
                (void) sprintf(
                               fdarg, filter_cmd_argv[filter_cmd_fd_info],
                                filter_cmd_fdout);
                filter_cmd_argv[filter_cmd_fd_info] = fdarg;
            }

            /*
             * Send real uid string on filter_cmd_fdout to client process
             */

            i = (int)strlen(human_name);
            if ((n = looprw(
              filter_cmd_fdout, human_name, i, write_CDL_no_eintr)) != i)
            {
                rbe_log_stats(RBRECOVER_MKERR(
                          errno), "Can't write %d bytes to host \"%s\", "
                              "ec=%d, errno=%d, fd=%d\n", i,rcmd_stuff[0],
                                                               n, errno,
                              filter_cmd_fdout);
                _exit(2);
            }

            if ((n = looprw(
                    filter_cmd_fdout, "\n", 1, write_CDL_no_eintr)) != 1)
            {
                rbe_log_stats(RBRECOVER_MKERR(errno),
                              "Can't write %d bytes to host \"%s\", "
                              "ec=%d, errno=%d, fd=%d\n",
                              1,rcmd_stuff[0],n,errno,filter_cmd_fdout);
                _exit(3);
            }

            (void)close(xcpiogen_pipe[0]);
           (void)dup2(xcpiogen_pipe[1], 2);
            (void)close(xcpiogen_pipe[1]);

            /* Call the CDL layer so we can warn SSL that an execvp
            is coming. If this isn't an SSL socket this call is a
            no-op
            */
            (void)CDL_execvpPrep(filter_cmd_fdout);

            (void)execvp(filter_cmd, filter_cmd_argv); /* run xcpiogen */
            rbe_log_stats(RBRECOVER_MKERR(
                                errno), "Can't exec \"%s\", errno=%d\n",
                          filter_cmd,errno);
            _exit(1);
            break;



        default:
             /* parent */
          (void)close(xcpiogen_pipe[1]);
          xcpiogen_prog_fd = xcpiogen_pipe[0];
          break;
        }   /* end of switch */

        /* Parent has no use for this file descriptor any more,
                                                          and nuking it
```

```
         * now is important so that if the child dies while the rcmd still
         * wants more input from the child, the rcmd will die too (
                                                                instead of
         * hanging around hoping that maybe this parent process will
                                                                    supply
         * the data) */

        /* we didn't really dup these if they were SFP sockets so
           avoid the closes in that case
        */
        if (!pwi->ssl_groupname)
        {
             if (rcmd_fd[0] != -1)
             {
                 (void)close(rcmd_fd[0]);
             }
             if (rcmd_fd[1] != -1)
             {
                 (void)close(rcmd_fd[1]);
             }
        }
        /* However in the case of nwrcmd we did get a network
          socket for  rcmd_stderr so we do need to issue one
          CDL_close on the one SFP socket. We can detect that
          case because rcmd_stderr will not be equal to rcmd_fd[0]
        */
        else
        {
             if (rcmd_stderr != rcmd_fd[0])
             {
                 if (rcmd_fd[0] != -1)
                 {
                  /*
                   * IF not a SSL socket, do the close if not
                   * we close SP socket in XCPIOGEN after
                   * data has been moved by calling STPexit()
                   */

                  if ( 0 == CDL_isSSLsocket(rcmd_fd[0]) )
                  {
                    (void) CDL_close(rcmd_fd[0]);
                  }

                  rcmd_fd[0] = -1;
                 }
             }
        }

        if ( (0 == strcmp(method, "rsh")) || (0 == strcmp(
                                                   method, "edmlink")) )
        {
         /* Because of occasional loss of data, due to timing issues when
          * connected via edmlink,
                                 turn on keepalive.  In version 4.7 this will
          * also be addressed by edmlink, directly. (OSGsw38596)
          */
            int on = 1;
            if (-1 == setsockopt( rcmd_stderr,
                                  SOL_SOCKET,
                                  SO_KEEPALIVE,
                                  (char *)&on,
                                  sizeof(on) ) )
            {
                rbe_log_stats( RBRECOVER_MKERR(errno),
                               "Warning: setsockopt for SO_KEEPALIVE
                                                                 failed" );
        }
    }

    if(0 == pkt0.failcode)
    {
      /* Lets check here to see if xcpiogen is still running.
       * The child could have exited prior to the fork.
       */
      int ChildDone_ret;
      int ChildExitStatus;

      r_resultdata = 0;          /* "meaningless" */

      xcpiogen_still_running = TRUE;
      sleep (1);
      ChildDone_ret = ChildDone(xcpiogen_pid, &ChildExitStatus);
      if(-1 == ChildDone_ret)
      {
         rbe_log_stats(RBRECOVER_MKERR(errno),
                       "Internal error: testing to see if xcpiogen
                                                            exited.");
      }
      else if (0 != ChildDone_ret)
      {
         xcpiogen_still_running = FALSE;
         r_resultdata = ChildExitStatus;
      }
    }
    /*
     * Send the setup failure/success packet.
     */

send_0:
    if (pkt0.msglen < 0)
    {
         pkt0.msglen = (int)strlen(pkt0_errstr) + 1;
    }

    c = '0';
    i = (int)sizeof(pkt0) + pkt0.msglen;
    pwrite_or_die(cxp->ap_w_fd, &c, 1, _exit);
    pwrite_or_die(cxp->ap_w_fd, (char *)&i, sizeof i, _exit);
    pwrite_or_die(cxp->ap_w_fd, (char *)&pkt0, sizeof pkt0, _exit);

    if (pkt0.msglen > 0)
    {
         sprintf(cxp->ap_error_message, "%s", pkt0_errstr);
         rbe_log_stats(0, "%s", cxp->ap_error_message);
         pwrite_or_die(cxp->ap_w_fd, pkt0_errstr, pkt0.msglen, _exit);
    }

    /* If the fork was successful,
                             wait for the remote exit status to come back
     * on stderr,
                  and send it to parent in an 'R' reply.  If the fork was
     * unsuccesful, there is no 'R' reply, and, furthermore,
                                                      the value in the
     * 'r' reply is meaningless. */


    if ((xcpiogen_pid > 0) &&
         (TRUE == xcpiogen_still_running))
    {
         int remote_exitinfo;
         int Demux_ret = 0;
         Demux_ret = DemuxAuxChildren(cxp->ap_w_prog_fd,
```

```
                                              rcmd_stderr,
                                              rcmd_stuff[0],
                                              xcpiogen_prog_fd,
                                              xcpiogen_pid,
                                              &remote_exitinfo);
         if(0 != Demux_ret)
         {
           /*      rbe_log_stats(
                           0, "Error monitoring Auxproc's children.");*/
           /* error_logging */
         }


         /*
          * notify the main program of remote success/failure
          */

         c = 'R';
         i = sizeof(remote_exitinfo);
         pwrite_or_die(cxp->ap_w_fd, &c, 1, _exit);
         pwrite_or_die(cxp->ap_w_fd, (char *)&i, sizeof i, _exit);
         pwrite_or_die(cxp->ap_w_fd, (
                                  char *)&remote_exitinfo, i, _exit);
#if 0
         rbe_log_stats(0, "Auxproc(%d) remote exit is %d.", getpid(),
                          remote_exitinfo);
#endif
         /*
          * Is our connection method anything BUT "netware"?
          */

         if (0 != strcmp(method, "netware"))
         {     /***  YES  ***/
             /* Now wait for the exit code of the filter (
                                                   local) process
              *
              * waitpid() may return with EINTR before the child
              * proc exits, in that case, we want to continue
              * with the wait. (fix for OSGswl5487)
              */

             while ((waitpid(xcpiogen_pid, &wait_result, 0) == -1) &&
                    (EINTR == errno))
             {
                 /* empty while loop */
             }

             /*
              * exited while loop either because the return value of
              * waitpid is NOT -1, or the errno is NOT EINTR
              */
         }
         else
         {     /*** IT'S NETWARE ***/
             /*
              *
              ************ HACK **** HACK **** HACK **** HACK
                                                 ************
              ************ HACK **** HACK **** HACK **** HACK
                                                 ************
              ************ HACK **** HACK **** HACK **** HACK
                                                 ************
              ************ HACK **** HACK **** HACK **** HACK
                                                 ************
              ************ HACK **** HACK **** HACK **** HACK
                                                 ************
                 *
                 *  This hack is basically for Netware.
                                          Netware TCP/IP close()
                 *  does not send anything back to the (
                                          local)filter process
                 *  that is running as pid pkt0.pid.
                                          So the "filter" process
                 *  just keeps on sending data across the network and
                                                         there isn't
                 *  anybody there to read it.
                 *
                 *  Therefore,
                        this "auxproc" doesn't finish until all data has
                 *  been written to the socket even though no process is
                                                             reading
                 *  it. However,
                       the process might hang indefinitely if the network
                 *  buffer(s) become(s) full.
                 *
                 *  This is how we will get around this (
                                          kind of a hack but could
                 *  not think of a better way).
                                         None of the normal TCP/IP
                 *  mechanisms seem to work with Netware.
                 *
                 *  If the exitinfo (
                                which is zero for success or non-zero for
                 *  failure) reports failure...
                 *
                 *  After we have received the exit info from the Netware
                                                              client
                 *  then try to get it's "wait_result" twice with a 2
                                                        second pause
                 *  in between.  If we don't get it,
                                          we have to assume that the
                 *  process is hung and can't wind down.
                 *
                 *  Therefore,
                           let's send a SIGPIPE to the process.  I chose
                 *  SIGPIPE because this is usually trapped by our
                                                        software and
                 *  handled in a controlled manner.
                                         Just keep on trying to kill
                 *  it, sleeping for 2 seconds,
                                       and then see if it went away until
                 *  it DOES go away.
                 *
                 *  Just keep on trying to kill it,
                                          sleeping for 2 seconds, and
                 *  then see if it went away until it DOES go away.
                 *
                 *  Appearently, the kill(
                                 SIGPIPE) is ignored by the process while
                 *  there is activity occuring like positioning the tape.
                                                             That is
                 *  the reason for having to do the kill(
                                                ) in a loop until it
                 *  works.
                 */

                if (remote_exitinfo)
                {
                    /**
                    ** We recieved an error exit status from Novell
                    **/
                    int exitinfo;
```

```
                    exitinfo = waitpid(
                                xcpiogen_pid, &wait_result, WNOHANG);
                    while (!exitinfo)
                    {
                        /*
                         * Wait a couple of seconds longer and try again
                         */

                        sleep(2);
                        exitinfo = waitpid(
                                xcpiogen_pid, &wait_result, WNOHANG);

                        if (!exitinfo)
                        {
                            kill(xcpiogen_pid, SIGPIPE);
                                                       /*  BLAST IT! */
                        }
                    }
                }
                else
                {
                    /**
                    ** We received a success return status from Novell
                    **/

                    /*
                     * waitpid() may return with EINTR before the child
                     * proc exits, in that case, we want to continue
                     * with the wait. (fix for OSGsw15487)
                     */

                    while ((waitpid(
                                xcpiogen_pid, &wait_result, 0) == -1) &&
                           (EINTR == errno))
                    {
                        /* empty while loop */
                    }
                    /*
                     * exited while loop either because the return value
                                                                      of
                     * waitpidis NOT -1, or the errno is NOT EINTR
                     */
                }
            }

            /*
             * main command loop in caller will send this result for us
             */

            /*
             * wait_result contains the exit status of the child
             * process. Use the standard macros to determine the
             * status of the child process and set the return value
             * to indicate the status accurately.
             */

            if (WIFEXITED(
                       wait_result))      /* child proc exited normally */
            {
                /*
                 * set the result to the exit code
                 */

                wait_result = WEXITSTATUS(wait_result);
            }
```

```
        else if (WIFSIGNALED(wait_result))
        {
            /*
             * child proc exited due to an uncaught signal, so
             * set the exit code to the signal number plus our own
             * code XG_EXIT_SIGBASE.
             */

            wait_result = WTERMSIG(wait_result) + XG_EXIT_SIGBASE;
        }
        else if (WIFSTOPPED(wait_result))
        {
            /*
             * child proc is stopped, set the exit code accordingly.
             */

            wait_result = XG_EXIT_STOPPED;
        }

        r_resultdata = wait_result;
    }

    if (rcmd_stderr != -1)
      {
        /* OSGsw34952 -- Workaround for STPclose() behavioral defect
         *
         * Shut off the SSL (STP) exit handler
         * IF not a SSL socket, do the close if not
         * we close SP socket in XCPIOGEN after
         * data has been moved by calling STPexit()
         */
       if ( 0 == CDL_isSSLsocket(rcmd_stderr) )
       {
         (void)close(rcmd_stderr);
         rcmd_stderr = -1;
       }
       else /* SSL socket used as stderr, so clean up
               the SP stuff */
       {
         /* SP socket was set to stderr of remote
            commands.  Since, we call CDL_noatexit
            when we created the SP socket we will
            explicitely call CDL_exit() that will
            clean up the entire SP environment */

          if ( 0 == sp_cdlexitdone )
          {
            CDL_exit();
            sp_cdlexitdone = 1;
          }
        }
      }

    /*
     * set up variables that main loop will use to send 'r' reply
     */

    cxp->ap_resultlen = sizeof r_resultdata;
    cxp->ap_resultdata = (char *)&r_resultdata;
}       /* end of z_rcmdfilter() */
```

```
static enum input_states decodecookie(char *cp);


struct cookie2state {
        char *cookie;
        enum input_states state;
};

static struct cookie2state c2stbl[] = {
        { REMFD_COOKIE_FD2OUT,          INSTATE_COPY_TO_STDOUT },
        { REMFD_COOKIE_FD2OUT2,
                                           INSTATE_COPY_TO_STDOUT },
        { REMFD_COOKIE_FD2OUTEND,       INSTATE_SEARCH_PREFIX0 },
        { REMFD_COOKIE_STATUS,          INSTATE_GATHER_STATUS },
        { NULL,                                 INSTATE_NG }
};


static enum input_states
decodecookie(char *cp)
{
    struct cookie2state *c2s;


    for (c2s = c2stbl; c2s->cookie != NULL; c2s++)
    {
        if (strcmp(c2s->cookie, cp) == 0)
        {
            return c2s->state;
        }
    }
    return INSTATE_SEARCH_PREFIX0;
}       /* end of decodecookie() */
```

```
static void
sigusr1_handler(int sigvalue)
{
    if (debugmode)
    {
      rbe_log_stats(0,"\nUSR1 (ATTN) signal received!\n");
    }

    if (0 < xcpiogen_pid)
    {
      (void)kill(xcpiogen_pid, SIGTERM);
    }

    ++attn_ec;
}       /* end of sigusr1_handler() */
```

```
static void
sigterm_handler(int sigvalue)
{
    if (debugmode)
    {
      rbe_log_stats(0,"\nTERM signal received!\n");
    }

    rbe_close_logs(logging_channel);

    /* ESS workaround: Make a call to CDL_exit
       if symmpath workitem and CDL_exit has
       not already been called.  This is to clean
       up sockets since we have made a call to
       CDL_noatexit, we have to make sure we
       clean up */

    if ( (1==is_symmpath) && (0==sp_cdlexitdone) )
    {
      CDL_exit();
      sp_cdlexitdone = 1;
    }

    signal(SIGTERM, SIG_DFL); /* restore default (terminate) action */
    (void)kill(getpid(), SIGTERM);

} /* end of sigattn_handler() */
```

```
/*
 * Invoked when running recover in debugging mode.
 * This execs a separate a.out file to implement the auxproc, so that
 * breakpoints can be set in the recover code without affecting
 * the auxproc code.
 */

static void
z_exec_separate_auxproc(struct auxproc_context *cxp,
                        char *pathname)
{
    char procnum_str[32];
    char r_fd_str[32];
    char w_fd_str[32];
    char r_bulk_fd_str[32];
    char dbgmodestr[32];
    char *argv0;

    /*
     * NOTE: This name is "special"; the recover main()
     * function looks for it to know when it's being invoked
     * to perform auxproc processing.
     */

    argv0 = "ebr_auxproc";

    (void)sprintf(procnum_str, "%d", cxp->ap_my_auxnum);
    (void)sprintf(r_fd_str, "%d", cxp->ap_r_fd);
    (void)sprintf(w_fd_str, "%d", cxp->ap_w_fd);
    (void)sprintf(r_bulk_fd_str, "%d", cxp->ap_r_bulk_fd);
    (void)sprintf(dbgmodestr, "%d", debugmode);
    (void)execlp(pathname,             /* prog to execute */
                 argv0,                        /* argv 0 */
                 procnum_str,          /* argv 1 */
                 r_fd_str,             /* argv 2 */
                 w_fd_str,             /* argv 3 */
                 r_bulk_fd_str,                /* argv 4 */
                 dbgmodestr,           /* argv 5 */
                 (char *)0);           /* end-of-args */

    /*
     * if we get here, the exec did not go. Caller will bomb for us.
     */

}       /* end of z_exec_separate_auxproc() */
```

```
/*
 * Use these functions with looprw to build
 * a loop read (or loop write) that ignores EINTR.
 */

static int
read_CDL_no_eintr(int fd,
                  char *buf,
                  int nbytes)
{
    int r;

    do
    {
        errno = 0;
        r = CDL_read(fd, buf, (uint_t)nbytes,0);
    } while (r == -1 && errno == EINTR);

    return r;
}       /* end of read_CDL_no_eintr() */
```

```
static int
write_CDL_no_eintr(int fd,
               char *buf,
               int nbytes)
{
    int r;

    do
    {
        errno = 0;
        r = CDL_write(fd, buf, (uint_t)nbytes,0);
    } while (r == -1 && errno == EINTR);

    return r;
}       /* end of write_CDL_no_eintr() */
```

```
/*
 * Wait, interruptibly,
                     for at least one byte to become available on fd.
 *
 * Returns 0 if at least one byte is available.
 * Returns -1 for any type of failure, including wait interruption.
 * Sets errno appropriate when -1 is returned.
 */

int
fd_avail_1_wait_intr(int fd)
{
    esl_fdset_ty rdbits;


    E_FD_ZERO(&rdbits);
    E_FD_SET(fd, &rdbits);

    /*
     * Don't need to examine rdbits after select, since only one
     * fd is in the set -- therefore return value can be computed
     * directly from select return value.
     */

    if (esl_select(E_FD_SETSIZE, &rdbits, NULL, NULL, NULL) == -1)
    {
        return -1;
    }
    return 0;
}       /* end of fd_avail_1_wait_intr() */
```

```
/*
 * Test, interruptibly,
                       for at least one byte to become available on fd.
 *
 * Returns 1 if at least one byte is available.
 * Returns -1 for any type of failure, including test interruption.
 * Return 0 if no data is available.
 *
 * Sets errno appropriate when -1 is returned.
 */

int
fd_avail_test_intr(int fd)
{

    int retStatus;
    struct pollfd fd_test;

    fd_test.fd = fd;
    fd_test.events = POLLIN;
    fd_test.revents = 0;


    /*
     * Don't need to examine rdbits after select, since only one
     * fd is in the set -- therefore return value can be computed
     * directly from select return value.
     */

    if ((retStatus = CDL_poll_read(&fd_test, 1, 0)) == -1)
    {
      /* ERROR encountered */
      return -1;
    }
    return retStatus;

}        /* end of fd_avail_test_intr() */
```

```
/*
 * Test, for at least one byte to become available on fd.
 *
 * Returns 1 if at least one byte is available.
 * Returns -1 for any type of failure other than EINTR.
 * Return 0 if no data is available.
 *
 * Sets errno appropriate when -1 is returned.
 */

int fd_avail_test1(int fd)
{
    int retStatus;
    struct pollfd fd_test;

    fd_test.fd = fd;
    fd_test.events = POLLIN;
    fd_test.revents = 0;


    /*
     * Don't need to examine rdbits after select, since only one
     * fd is in the set -- therefore return value can be computed
     * directly from select return value.
     */

    while ((-1 == (retStatus = CDL_poll_read(&fd_test, 1, 0))) &&
            (EINTR == errno))
    {
      ;
    }
    return retStatus;
}
```

```
#define EBR_FORK fork    /* portability definition */

static int ebr_direct_rcmd_fds[2] = { -1, -1 };

/*
 * Function to do a direct fork()/exec(
                                   ) instead of an rcmd to start up the
 * client programs.
            Returns ptr to in & out fds for stdi/o with client process.
 * Returns ptr to fds if successful,
                                   NULL if an error.  stderr from new
 * process is returned on *fd2p.
                             Should only be called when client == server.
 */

int *
ebr_direct_rcmd(char             **ahost,
                ushort_t           inport,
                struct auxproc_context *cxp,
                char               *locuser,
                char               *remuser,
                char               *cmd,
                int                *fd2p)
{
    int pid;
    int x[2];
    int y[2];
    int z[2];
    int fd_w;
    int fd_r;
    int fd_err;
    char *client_home = NULL;
    char *p;
    struct passwd *pw;
    struct sigaction    new_act;
    struct sigaction    old_act;
    int                 istat;


    /*
     * now open pipes that will become daemons stdin, stdout, stderr
     * remember that x[0] is for read and x[1] is for write
     */

    if ((-1 == pipe(x)) || (-1 == pipe(y)) || (-1 == pipe(z)))
    {
      WriteFmtStringMsg(
                    cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                        "Unable to create a pipe\n");
      return NULL;
    }
    fd_w   = x[1];      /* pipe for child's stdin */
    fd_r   = y[0];      /* pipe for child's stdout */
    fd_err = z[0];      /* pipe for child's stderr */

    /*
     * Prepare sigaction parameters
     * ignore SIGCHLD during this
     */

    sigemptyset(&new_act.sa_mask);
    new_act.sa_handler = SIG_IGN;
    new_act.sa_flags   = E_SA_RESTART;
    istat = sigaction(SIGCHLD, &new_act, &old_act);
```

```
    if ((pid = EBR_FORK()) == 0)
    {
        /*
         * child processing goes here while parent is stopped (
                                                          if vfork())
         * first, setup stdio to work using the pipes
         */

        if ((close(0)<0) || (close(1)<0) || (close(2)<0))
         {
          WriteFmtStringMsg(cxp->ap_w_prog_fd,
                            EDMREPROGMSG_AUXPROC_ERROR, 0,
                            "unable to close std(
                                in|out|err} for forked child\n");
          _exit(-1);
        }
        if ((dup(x[0])<0) || (dup(y[1])<0) || (dup(z[1])<0))
         {
          WriteFmtStringMsg(
                    cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                        "unable to dup pipe ends for forked
                                                          child\n");
          _exit(-1);
        }
        if ((close(x[0])<0) || (close(y[0])<0) || (close(z[0])<0)
           || (close(x[1])<0) || (close(y[1])<0) || (close(z[1])<0))
         {

          WriteFmtStringMsg(
                    cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                        "unable to close pipe ends for forked
                                                          child\n");
          _exit(-1);
        }

        /*
         * At this moment the process has a real UID of the user
                                                          executing
         * [x]ebrecover and an effective UID of root (
                                              since [x]ebrecover has
         * the setuid bit set and is owned by root).
         *
         * The first problem is the access(
                                      ) function call uses the real UID
         * of the process.  So if the user is not root or ebadmin,
                                                          the access
         * function will fail.  This causes the recovery to fail.
                                                               Thus
         * problem 1.
         *
         * The second problem is System V Unix does not pass the
                                                       effective UID
         * to processes doing an exec of a shell.
                                           Instead the effective UID
         * is changed back to the real UID of the process. However,
                                                                 BSD
         * implementation does maintain the effective UID of the
                                                            process.
         *
         * On System V,
               neither the real or effective UID will be root during
         * the exec of the shell script if the user executing
                                                     [x]ebrecover is
         * a non-root user.  This process will not be able to perform
         * /bin/sh -c .
                /<nodename>/bin/startrec because this processes real
         * and effective UID (which are now the same) does not have
         * permissions to the directory ~ebadmin.
         *
         * The recxcpio process will be reading the username of the
                                                          initiating
         * process from the standard input and setting the real UID to
                                                                 the
         * UID of that user so it will do no harm to set the real UID
                                                             to root
         * at this point.
                    It will be changed back to the actual user as one
         * of the first things in recxcpio.
         *
         */

        (void)setuid(geteuid());

        /*
         * now chdir to the client code home location
         */

      if (NULL == (pw = getpwnam(
                            remuser))) /* lookup client home dir */
        {
           p = "Can't get home directory";
           goto default_home;
        }
      else
        {
           if ((
      client_home = pw->pw_dir) != NULL) /* if there is a pointer */
             {
                if (0 != chdir(client_home))  /* cd to directory */
                  {
                     p = "Can't chdir to home";
                     goto default_home;
                  }
             }
             else
             {
                 p = "Can't figure out home dir";
default_home:
                 WriteFmtStringMsg(cxp->ap_w_prog_fd,
                                EDMREPROGMSG_AUXPROC_WARNING, 0,
                                "%s;
            \"%s\" for user \"%s\", defaulting to /usr/epoch/EB\n",
                                p, (
              client_home == NULL) ? "??" : client_home, remuser);
                 client_home = "/usr/epoch/EB/CLIENT_HOME";
             }
        }

       /*
         * client_home now points to the home dir of the client
                                                            software
         */

        /*
         *  Make sure that the home directory that we finally ended
                                                             up with
         *  is really there.
         *
         *  Yea.. I know.
                      I may have already done this if I found a passwd
         *  entry for the client backup username and it had a home
                                                          directory.
```

```
          *  However,
               I have to do it again just in case I came from another
          *  path.
              I did not really feel like trying to fix up the spagetti
          *  code.  I'm feeling a little lazy today...
          */

        if (0 != chdir(client_home))
        {
          WriteFmtStringMsg(
                    cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                            "Server backup directory \"%s\" not found
                                                or not searchable\n",
                               client_home);
          _exit(-1);
        }

       (void)execl("/bin/sh", "sh", "-c", cmd, 0);
       WriteFmtStringMsg(
                    cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                         "Unable to execl /bin/sh sh -c %s\n",cmd);
      _exit(127);
   }

   /*
    * parent code resumes here
    */

   /*
    * Close the parent's copies of the child-ends of the pipes
    */

   close(x[0]);                           /* pipe for child's stdin  */
   close(y[1]);                           /* pipe for child's stdout */
   close(z[1]);                           /* pipe for child's stderr */

   /*
    * check for fork() failure
    */

   if (-1 == pid)
   {
       WriteFmtStringMsg(
                    cxp->ap_w_prog_fd, EDMREPROGMSG_AUXPROC_ERROR, 0,
                          "fork() failed\n");

       close(x[1]);                       /* pipe for child's stdin  */
       close(y[0]);                       /* pipe for child's stdout */
       close(z[0]);                       /* pipe for child's stderr */
        if (istat == 0)
        {
           sigaction(SIGCHLD, &old_act, NULL);
        }
       return NULL;
   }

   /*
    * now fill in fd's to be returned
    */

   ebr_direct_rcmd_fds[0] = fd_w;      /* pipe for child's stdin */
   ebr_direct_rcmd_fds[1] = fd_r;      /* pipe for child's stdout */
   if (NULL != fd2p)
   {
       *fd2p = fd_err;
   }
```

```
    if (istat == 0)
    {
        sigaction(SIGCHLD, &old_act, NULL);
    }

    return ebr_direct_rcmd_fds;
}   /* end of ebr_direct_rcmd() */
```

```
/*
 * routine to search the clients_installed file for a clients
                                                             connection
 * method.  Returns a pointer to the connection method string if
                                                             successful,
 * NULL if the client entry could not be found.
 */

static char *
rb_getmethod(register char *host,
                uint_t *concurrency,
                uint_t *client_type)
{
    FILE                *file_ptr;
    FILE                *lock_ptr;
    FFDB_HOLDER         *inclient_holder;
    ffdb_inclient_ty    *installed_client;
    eperrno             errnum;
    static char         rb_ci_lastmethod[32] = "";

#ifdef PARAM_CHECK
    if (NULL == host)
    {
        rbe_internal_error(RBRECOVER_MKERR(
                                      EINVAL), null_arg, "host name");
        return NULL;
    }
#endif /* PARAM_CHECK */

    /*
     * get pathname to file
     */

    if (ffdb_inclient_path == NULL) /* if file name needs to be
                                                            prepared */
    {
        if (ffdb_inclient_init(FFDB_INIT_DEFAULT_PATH) != 0)
        {
            return NULL;
        }
    }

    /*
     * request was not in cache -- go load it
     */

    if ((lock_ptr = ffdb_lock_file(ffdb_inclient_path,
                                    "restore scanning
                                            clients_installed file",
                                    1, NULL, 0)) == NULL)
    {
        (void)rbe_user_error(
         ffdb_errnum, "Unable to obtain lock on \"%s\" for scanning",
                       ffdb_inclient_path);
        return NULL;
    }

    if ((file_ptr = ffdb_open_file(ffdb_inclient_path)) == NULL)
    {
        (void)rbe_user_error(
             ffdb_errnum, "Unable to open file \"%s\" for scanning",
                       ffdb_inclient_path);
        ffdb_unlock_file(lock_ptr);
        return NULL;
    }
    while ((inclient_holder = ffdb_read_record(
                                   file_ptr, &errnum)) != NULL)
    {
        installed_client = (ffdb_inclient_ty *)inclient_holder->drec;
        if (errnum == 0)
        {
            if (ebc_same_host(
                  host, installed_client->hostname)) /* a match? */
            {
                if (concurrency != NULL)
                {
                    if (0 == installed_client->concurrency)
                    {
                        *concurrency = 32767;
                    }
                    else
                    {
                        *concurrency = installed_client->concurrency;
                    }
                }
                if (client_type != NULL)
                {
                    *client_type = installed_client->platform;
                }
                strcpy(rb_ci_lastmethod, installed_client->method);
                (void)ffdb_close_file(file_ptr);
                ffdb_unlock_file(lock_ptr);
                ffdb_destroy_holder(inclient_holder);
                return rb_ci_lastmethod;
                                          /* return allocated string */
            }
        }
        ffdb_destroy_holder(inclient_holder);
    }

    /*
     * if we get here -- did not find host in clients_installed file
     */

    (void)ffdb_close_file(file_ptr);
    ffdb_unlock_file(lock_ptr);
    return NULL;
}   /* end of rb_getmethod() */
```

```
/*
 *  Reads remote file descriptor for reads of stderr
 *  of the remote cmd. Read is done until nothing is left
 *  then returns with or without exit code.
 *  The error/warning messages come first and a dealt with,
 *  if the exit code of the remote cmd is returned.
 *
 * (Inp) int fd -- The file descriptor to wait on and read.
 * (Out) int *exitp -- The Exit code.
 * (Inp) char *remhostname -- The remote client name.
 * (Inp) boolean_ty first_call -- Is this the first call ??
 * (I/O) enum input_states *state_ptr
 * (I/O) enum input_states *next_state_ptr
 * (I/O) boolean_ty *skipping_leading_whitespace.
 * (I/O) int *parsePos the parse position of the messages.
 * (I/O) int *msgPos the position of the logging message.
 *
 * None of the I/O vars need to be initialized for the first call.
 *
 * Returns: boolean_ty
 * False: The remote command has not exitted.
 * True: The remote command has exitted.
 */
static boolean_ty
parse_remote_stderr_info2(int prog_fd,
                          int remote_fd,
                          int *exitp,
                          char *remhostname,
                          boolean_ty first_call,
                          enum input_states *state_ptr,
                          enum input_states *next_state_ptr,
                          boolean_ty *skipping_leading_whitespace,
                          int *parsePos,
                          int *msgPos)

{
    enum input_states   previous_state;
    static char         cookiebuf[REMFD_MAX_COOKIE_LEN+1];
    int                 protfailed          = 0;
    int                 done                = 0;
    char                c                    = 0;
    int                 n;
#define MSGBUFFLEN 260
    static char         msgLogBuff[MSGBUFFLEN + 1];
    boolean_ty          ret_status              = FALSE;
    static int          temp_exit_status;
    int write_prog = 1;

    *exitp = 0; /* Initialize to success. */

    if (TRUE == first_call)
    {
      /* First time processing. */

      if (debugmode)
      {
         rbe_log_stats(
                   0, "AUXPROC: Processing stderr info from fd %d\n",
                       remote_fd);
      }

      *state_ptr = INSTATE_NG;
      *next_state_ptr = INSTATE_SEARCH_PREFIX0;
      *parsePos = 0;
      *exitp = 0;
      *msgPos = 0;
      temp_exit_status = 0;
      *skipping_leading_whitespace = FALSE;
      n = 0;
    }
    else
    {
      n = *parsePos;
    }

    while (!protfailed && !done)
    {
         int start_ec;
         int r;


         /*
          * Wait for the next character from the remote
          * stderr stream.  Ignore interrupts, unless
          * they were due to the attention signal.
          */

         start_ec = attn_ec;
         do
         {
             r = fd_avail_test_intr(remote_fd);
             if (0 == r)
             {
               return ret_status;
             }
         } while (r == -1 && errno == EINTR && attn_ec == start_ec);

         /*
          * now that there is a character, read it
          */
         c  = 0;

         if (r != -1)
         {
           do
           {
             r = CDL_read(remote_fd, &c, 1, 0);
           } while((-1 == r) && (errno == EINTR) && (
                                            attn_ec == start_ec));
         }

         if (r != 1)
         {
             *exitp = SPEXIT_REMOTE_STDERR_FAIL;
             protfailed = 1;
             ret_status = TRUE;
             break;
         }

         previous_state = *state_ptr;
         *state_ptr = *next_state_ptr;

         switch (*state_ptr)
         {
         case INSTATE_SEARCH_PREFIX0:
             if (debugmode)
             {
               rbe_log_stats(0, "AUXPROC: (
                                        %c) INSTATE_SEARCH_PREFIX0\n", c);
             }
```

```
             if (c == REMFD_MAGIC_PREFIX[0])
             {
                 *next_state_ptr = INSTATE_SEARCH_PREFIXN;
                 n = 1;
                 *parsePos = 1;
             }
             break;

         case INSTATE_SEARCH_PREFIXN:
             if (debugmode)
             {
               rbe_log_stats(0, "AUXPROC: (
                                        %c) INSTATE_SEARCH_PREFIXN\n", c);
             }

             if (c != REMFD_MAGIC_PREFIX[n])
             {
                 *next_state_ptr = INSTATE_SEARCH_PREFIX0;
             }
             else
             {
               n++;
               (*parsePos)++;
               if (n == REMFD_MAGIC_LENGTH)
               {
                 *next_state_ptr = INSTATE_GATHER_COOKIE;
                 n = 0;
                 *parsePos = 0;
               }
             }
             break;

         case INSTATE_GATHER_COOKIE:
             if (debugmode)
             {
               rbe_log_stats(0, "AUXPROC: (
                                        %c) INSTATE_GATHER_COOKIE\n", c);
             }

             if (c == REMFD_MAGIC_SUFFIX[0])
             {
                 /*
                  * Got the cookie
                  */

                 cookiebuf[n] = '\0';
                 *next_state_ptr = INSTATE_SEARCH_SUFFIXN;
                 n = 1;
                 *parsePos = 1;
             }
             else if (strchr(REMFD_COOKIE_CHARS, c) == NULL)
             {
                 /*
                  * We found a valid prefix (else we would
                  * not be here) but the cookie contains an
                  * illegal character.  Should not happen;
                  * there has been some protocol failure.
                  */

                 *exitp = SPEXIT_REMOTE_STDERR_FAIL;
                 ret_status = TRUE;
                 protfailed = 1;
             }
             else if (n == REMFD_MAX_COOKIE_LEN)
             {
                /*
                 * Should have seen the SUFFIX[0] by now.
                 */

                *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
                ret_status = TRUE;
                protfailed = 1;
            }
            else
            {
                /*
                 * another cookie character.
                 */

                cookiebuf[n] = c;
                n++;
                (*parsePos)++;
            }
            break;

        case INSTATE_SEARCH_SUFFIXN:
            if (debugmode)
            {
              rbe_log_stats(0, "AUXPROC: (
                                 %c) INSTATE_SEARCH_SUFFIXN\n", c);
            }

            if (c != REMFD_MAGIC_SUFFIX[n])
            {
                *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
                ret_status = TRUE;
                protfailed = 1;
            }
            else
            {
              n++;
              (*parsePos)++;
              if (n == REMFD_MAGIC_LENGTH)
              {
                *next_state_ptr = INSTATE_NEWLINE;
              }
            }
            break;

        case INSTATE_GATHER_STATUS:
            if (debugmode)
            {
               rbe_log_stats(0, "AUXPROC: (
                                  %c) INSTATE_GATHER_STATUS\n", c);
            }

            if (isdigit(c))
            {
                char xx[2];

                xx[0] = c;
                xx[1] = '\0';

                temp_exit_status *= 10;
                temp_exit_status += atoi(xx);
                n++;
                (*parsePos)++;
                *skipping_leading_whitespace = FALSE;

            }
            else if (c == '\n')
```

```
            {
                *exitp = temp_exit_status;
                ret_status = TRUE;
                done = 1;
            }
            else if (c == ' ' &&
                     ((n == 0) || *skipping_leading_whitespace ))
            {
              *skipping_leading_whitespace = TRUE;

                /*
                 * no-op, skip leading whitespace
                 */
            }
            else
            {
                *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
                ret_status = TRUE;
                protfailed = 1;
            }
            break;

        case INSTATE_COPY_TO_STDOUT:

            /*
             * Eventually, the protocol should include
             * an explicit length for this state, so
             * we can do large read/writes and so we will not
             * be vulnerable to confusion based on gunk being
             * copied to stdout.
             *
             * For now, just shove the characters at stdout
             * and stop as soon as we fix PREFIX[0]
             */

            if (c != REMFD_MAGIC_PREFIX[0])
            {
                /*
                 * build a buffer to write to the restore log
                 */
                if (('\n' == c) || ((*msgPos) >= MSGBUFFLEN))
                {
                  msgLogBuff[*msgPos] = '\0';
                  (void) rbe_log_stats(0,
                                       "%s: %s",
                                       remhostname,
                                       msgLogBuff);
                  if (write_prog)
                  {
                    WriteStringMsg(prog_fd,
                                   EDMREPROGMSG_RESTORE_RCMD_UNKNOWN,

                                   0,
                                   msgLogBuff);
                  }

                  memset(msgLogBuff, 0, MSGBUFFLEN + 1);
                  *msgPos = 0;
                }
                else
                {

                   msgLogBuff[*msgPos] = c;
                   (*msgPos)++;
                }
            }
            else
            {
                n = 1;
                *parsePos = 1;
                *next_state_ptr = INSTATE_SEARCH_PREFIXN;
            }
            break;

        case INSTATE_NEWLINE:
            if (c == '\n')
            {
                if (previous_state == INSTATE_SEARCH_SUFFIXN) {
                    *next_state_ptr = decodecookie(cookiebuf);

                    memset(cookiebuf, 0, REMFD_MAX_COOKIE_LEN+1);

                    n = 0;
                    *parsePos = 0;
                }
                else
                {
                    *next_state_ptr = INSTATE_SEARCH_PREFIX0;
                }
            }
            break;
        }   /* end of switch */
    }   /* end of while loop */

    return ret_status;

}       /* end of parse_remote_stderr_info2() */
```

```
/*
 * ForwardXcpiogenProgress()
 *  REP: Restore Engine Process, AP: auxproc, XC: xcpiogen.
 *  This routine read progress messages from XC and forwards
 *  then to REP. This routine should not block on a read().
 *
 * Args:
 *  int xcpiogen_prog_fd -- the progres channel from XC to AP.
 *  int restore_engine_prog_fd -- the progress channel from AP to REP.
 *
 * Returns:
 *  int -- the number of forwarded messages. -1 for an error.
 *
 * Side Effects:
 *  Read and writes file descpitors.
 */
static int
ForwardXcpiogenProgress(int xcpiogen_prog_fd,
                        int restore_engine_prog_fd,
                        boolean_ty *zero_byte_read)
{
  int bytes_skipped = 0;
  int read_ret, write_ret;
  int fd_test;
  int msg_count = 0;
  prog_chan_msg xcpiogen_msg;
  int debug_prog = 0;
  int write_prog = 1;

  *zero_byte_read = FALSE;

  memset(&xcpiogen_msg, 0, sizeof(prog_chan_msg));

  while(1 == (fd_test = fd_avail_test1(xcpiogen_prog_fd)))
  {
    read_ret =  ReadMsg(
                  xcpiogen_prog_fd, &xcpiogen_msg, &bytes_skipped);

    if(0 != bytes_skipped)
    {
      (void) rbe_log_stats(0,
                        "The progress channel from xcpiogen had %d
                                           extranous bytes.");

    }
    if(0 == read_ret)
    {
      *zero_byte_read = TRUE;
      break;
    }
    else if(-1 == read_ret)
    {
      break;
    }
    if(debug_prog)
    {
      (void) rbe_log_stats(0,
                        "%s: %s",
                        "debug progress",
                        DEBUG_PROG_MESSAGE_PEAK(&xcpiogen_msg));
    }
    if(write_prog)
    {
          write_ret =  WriteMsg(restore_engine_prog_fd, &xcpiogen_msg);
       }
       msg_count++;
       free(xcpiogen_msg.pcm_body);

       memset(&xcpiogen_msg, 0, sizeof(prog_chan_msg));
     }
     if(-1 == fd_test)
     {
       (void) rbe_log_stats(RBRECOVER_MKERR(errno),
                            "Encountered error testing xcpiogen file
                                                         descriptor.");
       return -1;
     }
     return msg_count;
}
```

```
/*
 * static int DemuxAuxChildren()
 *
 * This function handles IPC communications between the following
 * processes:  REP: Restore Engine Process, AP: auxproc, XC: xcpiogen,
 * and RC: remote command. The RC sends error and warning messages
 * tp AP. When the RP is finished, RC's exit status is sent last.
 * The remote exit status indicates that the restore is finished.
 * RP error and warning messages are logged and forward on as
 * progress messages to RP. At the same time XC will send progress
 * informaton to AP, and AP will forward them to REP. XC does its
 * own logging.
 *
 * Args:
 * int progress_fd -- (In) this file descriptor is from AP to REP.
 * int remote_fd --  (In) this file descriptor is from RC to AP.
 * char *remote_progname -- (In) this the RP executable name.
 * int xcpiogen_fd -- (In) this file descriptor is from XC to AP.
 * int xcpiogen_pid -- (In) the pid for XC.
 * int *remote_exit -- (Out) the remote exit interperated.
 *
 * REP: Restore Engine Process, AP: auxproc, XC: xcpiogen,
                                                 RC: remote command.
 *
 * Returns: ??
 *
 */

static int
DemuxAuxChildren(int progress_fd,
                 int remote_fd,
                 char *remote_progname,
                 int xcpiogen_fd,
                 int xcpiogen_pid,
                 int *remote_exit)
{

        int remote_exitinfo;
        int forward_ret;
        boolean_ty remote_exitted = FALSE;
        struct pollfd monitor_fds[2];
        int poll_ret;
        boolean_ty remote_first_call = TRUE;
        int logging_channel = 0;
        int start_ec = attn_ec; /* attn_ec global */
        boolean_ty xcpiogen_zero_byte_reads;
        int number_fds;
        /* The below  args are needed by and maintained by
         * parse_remote_stderr_info2(). No need to initialize
         * or test them, they simply maintain state information
         * for multiple invocations of parse_remote_stderr_info2().
         */

        enum input_states remote_state_ptr;
        enum input_states remote_next_state_ptr;
        boolean_ty skip_whitespace;
        int parsePos;
        int MsgPos;

        number_fds = 2;
        monitor_fds[0].fd = remote_fd;
        monitor_fds[0].events = POLLIN;
        monitor_fds[0].revents = 0;
```

```
          monitor_fds[1].fd = xcpiogen_fd;
          monitor_fds[1].events = POLLIN;
          monitor_fds[1].revents = 0;

          while((FALSE == remote_exitted) &&
                ((0 == ( poll_ret = CDL_poll_read(
                                monitor_fds, number_fds, 5000))) ||
                (poll_ret > 0) ||
                 ((EINTR == errno) && (-1 == poll_ret) && (
                                        start_ec != attn_ec))))
          {
            if (0 == poll_ret)
            {
              /* timed out */
              /* check for the attention signal and other periodic
                                                          checks. */
              if (start_ec != attn_ec)
              {
                /* Timeout on read, AND we were canceled (SIGUSR1) --
                   If this is an SSL socket read timeout, the client may
                   have gone away without us knowing it,
                                            so break out of the loop
                   without the remote exit status.
                                              Do this for all remote
                    remote process connections,
                                        since the xcpiogen to recxcpio
                    connection might be SSL,
                                   but not the remote_fd. In this case
                    the remote process will not know that xcpiogen is
                                                                 gone.
                 */

                *remote_exit = SPEXIT_REMOTE_NO_STATUS;
                                               /* leaving w/o status */

                rbe_user_error (0,
                               "No remote exit status received in 5
                                                seconds -- stopped "
                               "waiting for aborted restore's remote
                                                process to finish");
                 break;

              }
            }
            else if (-1 == poll_ret)
            {  /* eintr */
              /* log this here */
              /*rbe_log_stats(errno, "auxproc -- ");*/
            }
            else
            {
              if((POLLRDNORM & monitor_fds[1].revents) ||
                 (POLLRDBAND & monitor_fds[1].revents) ||
                 (POLLIN & monitor_fds[1].revents) ||
                 (POLLHUP & monitor_fds[1].revents))
              {
                /* Xcpiogen process is expected to send progress
                                                          formatted
                 * messages,
                      including warnings and errors along with restore
                 * progress over its stderr channel.
                                                We will check xcpiogen
                 * prior to the remote channel.
                                          This presumes that the remote
                 * channel determines whether we are done with the
                                                              restore.
```

```
                 */
                monitor_fds[1].revents = 0;

                xcpiogen_zero_byte_reads = FALSE;

                forward_ret = ForwardXcpiogenProgress(
                                                    xcpiogen_prog_fd,
                                                     progress_fd,

                                           &xcpiogen_zero_byte_reads);
                if(TRUE == xcpiogen_zero_byte_reads)
                {
                  /* Lets no longer wait on this fd,
                   * if we get a zero byte read. The
                   * second fd is for xcpiogen.
                   */
                  number_fds = 1;
                }
              }
              if((POLLRDNORM & monitor_fds[0].revents) ||
                 (POLLRDBAND & monitor_fds[0].revents) ||
                 (POLLIN & monitor_fds[0].revents) ||
                 (POLLHUP & monitor_fds[0].revents))
              {
                /* POLLRDBAND Why do we need these conditions. */

                /* Remote process is expected to send information back
                                                            on stderr
                 * stream.  Read that information, parse it,
                                                      and if available
                 * send remote exit status back to parent.
                                                   If the exit status
                 * is read this loop is terminated.
                 */
                monitor_fds[0].revents = 0;

                remote_exitted = parse_remote_stderr_info2(progress_fd,
                                                          remote_fd,

                                                       &remote_exitinfo,

                                                         remote_progname,

                                                       remote_first_call,

                                                       &remote_state_ptr,

                                                  &remote_next_state_ptr,

                                                       &skip_whitespace,
                                                             &parsePos,
                                                             &MsgPos);
                remote_first_call = FALSE;

              }
            }
          }

          /* Should we test xcpiogen for a last time before we return!
                                                                    */

          if(1 == fd_avail_test(xcpiogen_prog_fd))
          {
            xcpiogen_zero_byte_reads = FALSE;

            forward_ret = ForwardXcpiogenProgress(xcpiogen_prog_fd,
                                                   progress_fd,

                                             &xcpiogen_zero_byte_reads);
         }

         if (-1 == poll_ret)
         {  /* Not eintr */
           rbe_log_stats(RBRECOVER_MKERR(errno),
                          "Auxproc failed to poll Children pids!");
           return -1;
         }
         if(TRUE == remote_exitted)
         {
           *remote_exit = remote_exitinfo;
           return 0;
         }

} /* DemuxAuxChildren() */
```

D12

```
/*********************************************************************
                                                               ******
**
** File Name:   RSLinitfin.c
**
** Copyright (c) 1998,1999 by EMC Corporation.
**
** Purpose:
**               This module contains the Restore Service Library
                                                        functions to
**               initialize and terminate the restore operation.
**
** Table of Contents:
** ------------------
**               RSTSL_Initialize
**               RSTSL_Finish
**
**        Internal Functions:
**
**
** Compile-Time Options:
**               This section must list any compile time definitions
**               which will affect this header.
**
*********************************************************************
                                                              *****/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1


/*
 * System headers.
 */
#include <sys/param.h>
#include <dirent.h>
#include <dlfcn.h>


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>
```

```
/*
 * Local headers
 */
#include <RSLinterns.h>
#include <RSLstartup.h>


/*
 * #defines, structures, typedefs local to this source file
 */

static eerrno_ty init_plugins( restore_context *rcp );
static int validate_plugin( struct pluginData *piDataPtr );


/*
 * External declarations
 *
 * This is the global "restore context" that will be used
 * throughout the rest of the restore operations.
 */
struct restore_context *rcp = NULL;


/*
 * Definitions of the names of the plugin functions in the piFuncArray
 * of the pluginData structure.
                                These must be in the same order and position
 * as the PIFuncIndex values defined in RSLplugin.h.
 */

char *piFuncNames[PIFuncIndexLast+1] = {
        "RSTPI_Identify",
        "RSTPI_Initialize",
        "RSTPI_GetTopLevelObjects",
        "RSTPI_SetTopLevelObject",
        "RSTPI_GetNextLevelObjects",
        "RSTPI_ClearRestoreContext",
        "RSTPI_Submit",
        "RSTPI_GetTopLevelTemplates",
        "RSTPI_DoesAlternateExist",
        "RSTPI_MarkObject",
        "RSTPI_UnmarkObject",
        "RSTPI_IsObjectMarked",
        "RSTPI_IsObjectMarkable",
        "RSTPI_GetAllBackupTimes",
        "RSTPI_GetCurrentBackupTime",
        "RSTPI_SetBackupForTime",
        "RSTPI_SetPrevBackup",
        "RSTPI_SetNextBackup",
        "RSTPI_SetFirstBackup",
        "RSTPI_SetMostRecentBackup",
        "RSTPI_IsTherePrevBackup",
        "RSTPI_IsThereNextBackup",
        "RSTPI_IsTherePrevBackupForTime",
        "RSTPI_IsThereNextBackupForTime",
        "RSTPI_Finish",
        "RSTPI_StartRestore",
        "RSTPI_FindRestorableObjects",
        "RSTPI_GetFindResults",
        "RSTPI_GetNecessaryMedia"
 };
```

```
/***********************************************************************
 * RSTSL_Initialize:
 *
 *  This function takes care of all the initialization for a restore
 *  session.  This must be called prior to any of the other functions
 *  in the Restore API.
 *
 * Parameters:
 *
 *  userName (I) - The name of the user.
 *
 **********************************************************************/

eerrno_ty
RSTSL_Initialize( const char *userName )
{
    eerrno_ty status = E_SUCCESS;

    /*
     * If we have not yet allocated space for a restore_context
     * structure, do so now.  If we have already done so,
                                                         just clear it
     * now.
     */

    if (NULL == rcp)
    {
        rcp = (struct restore_context *)malloc(sizeof(
                                           struct restore_context));
        if (NULL == rcp)
         {
             rec_api_log_csm(SUB_CSM_NOMEM, NULL);
             return(EP_RB_RECOVER_NOMEM);
         }
    }
    memset(rcp, 0, sizeof(struct restore_context));

    rcp->rc_human_uidname = esl_strdup( userName );


    if (!rcp->rc_human_uidname) {
      rec_api_log_csm(SUB_CSM_NOMEM, NULL);
      return(EP_RB_RECOVER_NOMEM);
    }

    /*
     * Set the appropriate field in the recovery context to indicate
     * that this recover session is based on the Recover API.
     * This flag is in place for historical reasons but is used by
     * other parts of the Recover API library.
     */

    rcp->gui_mode = 1;

    /*
     * Initialize the logging mechanism.
     */

    if (status = rbrlog_begin(rcp, progname))
    {
        return(status);
    }

    /*
     * Initialize the few "recover context" variables that we can at
     * this early stage.
```

```
     */

    setup_proc(rcp);

    /*
     * The following call will:
     *   -Initialize the saveset datatbase.
     *   -Infer any information we can at this point.
     */

    if (status = startup(rcp))
    {
        return(status);
    }

    /* Do plugins setup: Find and initialize all valid restore plugin
                                                               libs: */

    status = init_plugins( rcp );

    return( status );
}       /* End of RSTSL_Initialize() */
```

```
/************************************************************************
 * RSTSL_Finish
 *
 *   Function Description:
 *
 *  This function terminates a restoral session,
                                        but not while a restore is in
 *  progress.
        It will be rejected if a restore is currently being executed.
 *  This routine will clean up any local memory used in the session.
 *
 * Parameters:
 *
 *  none
 *
 */

eerrno_ty
RSTSL_Finish( void )
{
    int mc_n;


    eerrno_ty err = E_SUCCESS;

    if (NULL == rcp)
    {
        return( E_SUCCESS );
    }
    RemoveSubmitFiles();
    /*
     * Call rbr_cleanup() which kills the aux proc(
                                            s), unlocks the work
     * item, then calls rbrlog_end(
                                ) to enter the last logs and to close
     * the log file.
     */

    rbr_cleanup(rcp);

    /*
     * Deallocate the memory of restore_context and the related
     * structures.
     */

    if (NULL != rcp->rc_mcp)      /* Free the multicat structures */
    {
        mcat_destroy(rcp->rc_mcp);
    }

    /*
     * Free the mark bit map space
     */

    for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
    {
        if (NULL != rcp->rc_marks[mc_n])
        {
            free(rcp->rc_marks[mc_n]);
        }
        rcp->rc_marks[mc_n] = NULL;
    }

    if (NULL != rcp->rc_nmarks_by_plane)
```

```
    {
        free(rcp->rc_nmarks_by_plane);
    }

    /*
     * Free the configuration structures
     */

#if 0
    if (NULL != rcp->rc_cfgname)
    {
        free(rcp->rc_cfgname);
    }
#endif

    if (NULL != rcp->rc_config)
    {
        rbc_freeconfig(rcp->rc_config);
    }

    /*
     * Free the DS_NONE structures array
     * Note that even though rc_dsnones is the head of linked list
     * of dsnone_info structures, the list is allocated via malloc
     * as an array initially (ref. alloc_plane_arrays()), therefore
     * we can do a free here.
     */

    if (NULL != rcp->rc_dsnones)
    {
        free(rcp->rc_dsnones);
    }

    /*
     * Free the volume list structures.
     */

    if (NULL != rcp->ebvllist)
    {
        (void)ebvl_volidlist_destructor(
                                    rcp->ebvllist, EBVL_DESTROY_ALL);
    }

    /*
     * Free the plugin related data
     */

    rcp->rc_backup_app = 0;
    while (rcp->currentPIptr = rcp->piList)
    {
        rcp->rc_backup_app++;
        rcp->appData = rcp->currentPIptr->appData;
        /* allow plugin to clean up and close .so: */
        if ( E_SUCCESS != (err =
            rcp-> currentPIptr-> piFuncArray[ PIFuncIndexFinish ] (
                                                          rcp) ) )
        {
        /* log error, continue */
            rbe_user_error( err,
                        "RSTPI_Finish failed for restore plug-in
                                                library %s\n",
                ((struct pluginIDdata *)(
                        rcp-> currentPIptr-> idData))-> name );
        }
        dlclose( rcp-> currentPIptr-> libHdl );
        rcp->piList = rcp->piList->next;
```

```
            free (rcp->currentPIptr);
        }

        /*
         * Free the various simple string buffers
         */

        if (NULL != rcp->rc_top_level_object_name)
        {
            free(rcp->rc_top_level_object_name);
        }

        if (NULL != rcp->rc_template_name)
        {
            free(rcp->rc_template_name);
        }

        if (NULL != rcp->rc_workitem_name)
        {
            free(rcp->rc_workitem_name);
        }

        if (NULL != rcp->rc_human_uidname)
        {
            free(rcp->rc_human_uidname);
        }

        if (NULL != rcp->rc_effective_uidname)
        {
            /* don't free, its internal: free(rcp->rc_effective_uidname);
                                                                        */
        }

        if (NULL != rcp->rc_client_rbuname)
        {
            free(rcp->rc_client_rbuname);
        }

        if (NULL != rcp->rc_client_hostname)
        {
            free(rcp->rc_client_hostname);
        }

        if (NULL != rcp->rc_client_scriptname)
        {
            /* don't free, its internal: free(rcp->rc_client_scriptname);
                                                                        */
        }

        if (NULL != rcp->rc_client_dirtop)
        {
            free(rcp->rc_client_dirtop);
        }

        if (NULL != rcp->rc_cmd_context)
        {
            /* don't free -- its internal/temp data: free(
                                          rcp->rc_cmd_context); */
        }

        if (NULL != rcp->rc_source_client_hostname)
        {
            free(rcp->rc_source_client_hostname);
        }

        if (NULL != rcp->rc_cpiogen_executable)
```

```
        {
            /* don't free, its internal: free(rcp->rc_cpiogen_executable);
                                                                         */
        }

        if (NULL != rcp->rc_plugin_wi_types)
        {
            free(rcp->rc_plugin_wi_types);
        }

        if (NULL != rcp->rc_pwd)
        {
            free(rcp->rc_pwd);
        }

        /*
         * Finally, deallocate the restore_context itself
         */

        free(rcp);
        rcp = NULL;



        return( err );
    }       /* RSTSL_Finish */
```

```
/*********************************************************************
 * init_plugins
 *
 *   Function Description:
 *
 *  This function locates, opens,
                         validates and initializes all restore
 *  plug-in (shared) libraries.  They must be located in
 *  /usr/epoch/EB/cure_plugin (
                         eb_cure_plugin_dir).  All .so files in that
 *  directory are opened and validates for version# and presence of
                                                             all
 *  mandatory functions.
                     The RSTPI_Identify function is called for each
 *  library to determine which optional features are supported,
                                                          and that
 *  the corresponding functions are present.  Finally,
                                                the RSTPI_Initialize
 *  function is called for each valid library.
 *
 * Parameters:
 *
 * Inputs:
 *      rcp      (I)     - Pointer to restore context
 *
 * Outputs:
 *      none
 *
 * Returns:
 *      E_SUCCESS or EP_RB_RECOVER_xxx
 *
 * Logic/pseudo code:
 *
 *      open plugin dir
 *      while read_next_entry succeeds
 *              verify .so file (else continue)
 *              open shared library file (else continue)
 *              on errors below:
 *                      close shared library file
 *                      continue
 *              fetch all mandatory function addresses
 *              call Identify function
 *              validate version number
 *              fetch all indicated optional function addrs
 *              call Initialize function
 *              add workitem types to composite exclusion list
 *              add to valid plugin list
 *      close plugin dir
 */

static eerrno_ty init_plugins( restore_context *rcp )
{
        DIR                 *dirp;
        struct dirent       *direntp;
        eerrno_ty           status = E_SUCCESS;
        struct pluginData *piDataPtr = NULL;
        struct pluginData *piListPtr = NULL;
        int                 val_result;
        struct pluginIDdata *idDataPtr;
        char                *tmp_types;
        int                 shlib_dirlen;
        char                shlib_path [MAXPATHLEN];
```

```
        /* open plugin directory or bust */
        if ( NULL == (dirp = opendir( eb_cure_plugin_dir )) )
        {
                rec_api_log_csm( SUB_CSM_PLUGIN_ERR, NULL );
#if 1
                return E_SUCCESS;
                                /* allow continuation w/o plugins */
#else
                return EP_RB_RECOVER_NO_PLUGINS;
                                                  /* later do this */
#endif
        }

        strcpy( shlib_path, eb_cure_plugin_dir );
        strcat( shlib_path, "/" );
        shlib_dirlen = strlen (shlib_path);

        /* loop thru entries in directory*/
        while (NULL != (direntp = readdir( dirp )))
        {
                if (NULL == piDataPtr)
                {       /* allocate next plugin data structure */
                        if (NULL == (piDataPtr
                                   = calloc( 1, sizeof(
                                         struct pluginData) )))
                        {
                                status = EP_RB_RECOVER_NOMEM;
                                break;
                                        /* fall thru to cleanup */
                        }
                }

                if (NULL == strstr( direntp->d_name, ".so" ) )
                        continue;       /* skip this guy */

                strcpy( &shlib_path[shlib_dirlen], direntp->d_name );
                if (NULL == (piDataPtr->libHdl
                                = dlopen( shlib_path, RTLD_NOW )))
                {
                        rbe_user_error( 0,
                            "Error opening restore plug-in library
                                                         %s: %s\n",
                                    direntp->d_name, dlerror() );
                        continue;       /* skip this one */
                }

        /* Fetch addresses of all mandatory functions and */
        /* Do Identify processing: call it, save options, validate */

                if (0 != (val_result = validate_plugin(
                                                     piDataPtr ) ))
                {
                    if (val_result == -1 || val_result == -4)
                    {
                        rbe_user_error( 0,
                "Functions missing from restore plug-in library %s:
                                                             %s\n",
                                    direntp->d_name, dlerror() );
                    }
                    else if (val_result < 0)
                    {
                        rbe_user_error( 0,
                          "Validation failed for restore plug-in
                                                  library %s\n",
                                    direntp->d_name );
                    }
                else
                {
                    rbe_user_error( val_result,
                "RSTPI_Identify failed for restore plug-in library
                                                                %s\n",
                                    direntp->d_name );
                }

                dlclose( piDataPtr->libHdl );
                                        /* close .so on errors */
                piDataPtr->libHdl = NULL;
                continue;   /* on any error, skip this lib */
            }

        /* let DC plug-in do its initialization */
            rcp->appData = NULL;
                                /* enter plugin with clean appdata */
            status =
            piDataPtr->piFuncArray[PIFuncIndexInitialize]( rcp );
            if (E_SUCCESS != status)
            {
                    rbe_user_error( status,
            "RSTPI_Initialize failed for restore plug-in library
                                                                %s\n",
                                    direntp->d_name );
                    dlclose( piDataPtr->libHdl );
                                        /* close .so on errors */
                    piDataPtr->libHdl = NULL;
                    status = E_SUCCESS;
                                            /* this was't fatal */
                    continue;
                                    /* on any error, skip this lib */
            }

        /* save plugin's appData */
            piDataPtr->appData = rcp->appData;
            rcp->appData = NULL;

        /* add piDataPtr to valid plugin list */
            if (NULL == piListPtr)
                    rcp->piList= piDataPtr;
                                            /* first in list */
            else
                    piListPtr->next = piDataPtr;
                                            /* link from prev */
            piListPtr = piDataPtr;
                                            /* new end of list */
            piDataPtr = NULL;

        /* add workitem types to composite exclusion list */
            idDataPtr = (struct pluginIDdata *)piListPtr->idData;
            if (idDataPtr->num_types > 0)
            {
                tmp_types = calloc( 1, 1 + idDataPtr->num_types
                                + rcp->rc_num_plugin_wi_types
                                                                );
                if (NULL == tmp_types) {
                    status = EP_RB_RECOVER_NOMEM;
                    break;
                }
                if (NULL != rcp->rc_plugin_wi_types)
                {
              /* move old list to new buffer and free old list */
                    memcpy( tmp_types,
                            rcp->rc_plugin_wi_types,
                            rcp->rc_num_plugin_wi_types );
```

```
                    free( rcp->rc_plugin_wi_types );
                }
                memcpy( tmp_types + rcp->rc_num_plugin_wi_types,
                        idDataPtr->wi_types,
                        idDataPtr->num_types );
                rcp->rc_num_plugin_wi_types +=
                                        idDataPtr->num_types;
                tmp_types[rcp->rc_num_plugin_wi_types] = 0;
                rcp->rc_plugin_wi_types = tmp_types;
            }
        }

        (void)closedir( dirp );

        /* free up leftovers: */
        if (NULL != piDataPtr)
                free (piDataPtr);

        if (E_SUCCESS != status)
        { /* Free contents of plugin list: */
                while (NULL != (piDataPtr = piListPtr))
                { /* allow plugin to clean up and close .so: */
                        rcp->appData = piDataPtr->appData;
                        piDataPtr->piFuncArray[PIFuncIndexFinish](
                                                                rcp );
                        dlclose( piDataPtr->libHdl );
                        piListPtr = piDataPtr->next;
                        free (piDataPtr);
                }
        }

        return status;
}
```

```
        /* init_plugins */

/*********************************************************************
 * validate_plugin
 *
 *    Function Description:
 *
 *  This function retrieves the addresses of the mandatory plugin
                                                            functions
 *  and stores them in the function pointer array.
                                              If any function is missing
 *  it returns -1.
        It then calls the identify function and verifies wthe plugin
 *  version,
          and finds its optional functions. Specific error values are
 *  returned on version mismatch and missing optional functions.
 *
 * Parameters:
 *
 * Inputs:
 *      piDataPtr (
          I)    - Pointer to plugin data structure with libHdl set
 *
 * Outputs:
 *      piFuncArray in piDataPtr is loaded with pointers to plugin
                                                            functions
 *
 * Returns:
 *      0 on success
 *      -1 on any missing required functions
 *      -2 if version validation fails OR identify returns junk
 *      -3 if workitem type validation fails
 *      -4 on any missing optional functions indicated by options
                                                                flags
 *      +n (
   EB_RB_RECOVER_xxx) for error codes returned from Identify function
 *
 *********************************************************************
                                                                 ****/

static int validate_plugin( struct pluginData *piDataPtr )
{
        int             index;
        eerrno_ty       status;
        struct pluginIDdata *idDataPtr;

        for( index = 0; index <= PIFuncIndexLastBasic; index++ )
        {
                if (NULL == (piDataPtr->piFuncArray[index]
                        = (piFuncPtr) dlsym( piDataPtr->libHdl,
                                        piFuncNames[index]
                                                            ))
                )
                        return -1;
        }

        /* call identify and validate: */
        status = piDataPtr->piFuncArray[PIFuncIndexIdentify](

                                                &piDataPtr->idData );
        if (status != E_SUCCESS)
                return status;
        if (NULL == (idDataPtr = (
                        struct pluginIDdata *)piDataPtr->idData) )
                return -2;
```

```
        if (idDataPtr->version != RSTPI_VERSION)
        { /* only version 1 supported so far */
                piDataPtr->idData = NULL;
                return -2;
        }

        if (idDataPtr->num_types && !idDataPtr->wi_types)
        {       /* count cant be positive with null pointer */
               piDataPtr->idData = NULL;
               return -3;
        }

        /* if startRestore option set, get its addr or bust */
        if ( ( (RSTPI_OPTION_SPECIAL_START
                == (idDataPtr->options & RSTPI_OPTION_MASK_START))
            && (NULL == (
                        piDataPtr->piFuncArray[PIFuncIndexStartRestore]
                           = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                piFuncNames[PIFuncIndexStartRestore] )))
           )
        /* OR if special find option set, get its addr or bust */
          || ( (RSTPI_OPTION_SPECIAL_FIND
                == (idDataPtr->options & RSTPI_OPTION_MASK_FIND))
            && ( (NULL == (piDataPtr->piFuncArray[PIFuncIndexFind]
                           = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                        piFuncNames[PIFuncIndexFind] )))
              || (NULL == (
                         piDataPtr->piFuncArray[PIFuncIndexFindResults]
                           = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                    piFuncNames[PIFuncIndexFindResults])
           ) ) )            )
        /* OR if special getMedia option set, get its addr or bust */
          || ( (RSTPI_OPTION_SPECIAL_GET_MEDIA
                == (
                  idDataPtr->options & RSTPI_OPTION_MASK_GET_MEDIA))
            && (NULL == (piDataPtr->piFuncArray[PIFuncIndexGetMedia]
                           = (piFuncPtr) dlsym( piDataPtr->libHdl,

                                    piFuncNames[PIFuncIndexGetMedia] )))
        )     )
        {
                piDataPtr->idData = NULL;
                return -4;
        }

        return 0;
}
```

```
/*      validate_plugin */
```

```
/************************************************************************
******
**
** File Name:   RSLplugin.h
**
** Copyright (c) 1999 by EMC Corporation.
**
** Purpose:
**       Two-fold:
**               - Define prototypes of functions in a restore plugin
                                                               library
**               - Define prototypes of common functions exported from
                                                                   the
**              restore engine for use by restore plugin libraries.
                                                                 These
**              functions are part of the restore_legacy library.
**
**       This header defines the API that a backup application must
                                                             supply to
**       be compatible with the new Restore API.
                                              The Restore API provides
**       client-server based restore functionality formerly provided
                                                                by the
**       RECOVER_API.  The plug-in API is provided in a shared library,
                                                              designed
**       to be called by the Restore Service Library -- the generic
                                                           server side
**       component of the Restore API that executes as part of the
                                                               Restore
**       Engine (server).
**
** Compile-Time Options:
**       Need:
**               #define  _POSIX_SOURCE 1
**       Need header files:
**               <ebutil/ebutil.h>
**               <restore/restore_api.h>
**               <restore/restoreRPC.h>
**
*************************************************************************
*****/

#ifndef H_RSLPIAPI
#define H_RSLPIAPI

#ifdef __cplusplus
       extern "C" {
#endif

#include <ebutil/ebutil.h>
#include <eerrno/e_eb.h>
#include <restore/restore_api.h>
#include <restore/restoreRPC.h>

/* The following definition identifies the version of the Restore
                                                                Plugin
 * header that a plug-in library was built with.
                                            Whenever a plugin function
 * prototype changes, or a plugin function is added,
                                                  this value should be
 * updated.  It will be used by the restore engine code to verify
 * compatibility with individual restore plugin libraries.
 */
```

```
#define RSTPI_VERSION           1

/* Definitions of the indices of the plugin function addrs in the
                                                           piFuncArray
 * of the pluginData structure.
 */

/* Mandatory functions: */
#define PIFuncIndexIdentify     0
#define PIFuncIndexInitialize   1
#define PIFuncIndexGetTLO       2
#define PIFuncIndexSetTLO       3
#define PIFuncIndexGetNLO       4
#define PIFuncIndexClearRC      5
#define PIFuncIndexSubmit       6
#define PIFuncIndexGetTLTemps   7
#define PIFuncIndexDoesAltExst  8
#define PIFuncIndexMark         9
#define PIFuncIndexUnmark       10
#define PIFuncIndexIsObMarked   11
#define PIFuncIndexIsObMarkable 12
#define PIFuncIndexGetAllTimes  13
#define PIFuncIndexGetCurTime   14
#define PIFuncIndexSetBkupTime  15
#define PIFuncIndexSetPrevTime  16
#define PIFuncIndexSetNextTime  17
#define PIFuncIndexSetFirstTime 18
#define PIFuncIndexSetMostRcnt  19
#define PIFuncIndexIsPrevBkup   20
#define PIFuncIndexIsNextBkup   21
#define PIFuncIndexIsPrvForTime 22
#define PIFuncIndexIsNxtForTime 23
#define PIFuncIndexFinish       24
#define PIFuncIndexLastBasic     24

/* Optional functions: */
#define PIFuncIndexStartRestore 25
#define PIFuncIndexFind         26
#define PIFuncIndexFindResults  27
#define PIFuncIndexGetMedia     28
#define PIFuncIndexLast         28

/* Definitions of the names of the plugin functions in the piFuncArray
 * of the pluginData structure.
                                These must be in the same order and position
 * as the PIFuncIndex values above,
                                       and are initialized in RSLinitfin.c.
 */

char *piFuncNames[PIFuncIndexLast+1];

typedef eerrno_ty (*piFuncPtr)();
                                     /* generic plugin func prototype */

typedef struct restore_context restore_context;
                                            /* for function prototypes */

/* callback function prototypes: */

typedef boolean_ty  (*RSTPI_MarkProgressProc)(
                                            unsigned long totalMarks);
typedef boolean_ty  (*RSTPI_SubmitProgressProc)(
                                          unsigned long totalElements);
```

```
typedef boolean_ty  (*RSTPI_FindProgressProc)(
                                   unsigned long progressCount);
typedef boolean_ty  (*RSTPI_QuitTest)( void );

/** Definitions of the data that the RSTPI_Identify returns. **/

struct pluginIDdata {
        u_long version;
                              /* Version of the plugin header that the
                                   library was built with */
        char *name;
                             /* Name of the backup application (64 byte
                                   buffer address) */
        u_long options;
                           /* Bit mask identifying the optional plug-in
                                   features supported.
                                                  The bit definitions for
                                   this parameter (
                                         RSTPI_OPTION_MASK...) are
                                   defined below. */
        char *wi_types;
                        /* pointer to array of workitem types supported
                                   by the plugin */
        u_short num_types;
                           /* Number of witypes in the witypes array */
};

/** Definitions of the options bits in the RSTPI_Identify function
                                                          output: **/
/* NOTE: The output structure containing this data is defined in
                                                     RSLcontext.h */

/* This bit indicates whether the plug-in supplies its own function to
 * execute the actual restore.
 */
#define RSTPI_OPTION_MASK_START                     0x1
#define RSTPI_OPTION_SPECIAL_START                  0x1
#define RSTPI_OPTION_GENERIC_START                  0x0

/* The following bits indicate whether and how the plug-in supports
                                                              the find
 * function for searching its catalogs for specific objects.
                                                         The first bit
 * indicates if find is supported,
                      the second bit indicates whether the plugin
 * supplies its own find function,
                             or that the generic find function that
 * searches the mcat data can be used.
 */
#define RSTPI_OPTION_MASK_FIND                      0x6
#define RSTPI_OPTION_FIND_SUPPORTED                 0x4
#define RSTPI_OPTION_SPECIAL_FIND                   0x6
#define RSTPI_OPTION_GENERIC_FIND                   0x4
#define RSTPI_OPTION_NO_FIND                        0x0

/* The following flag indicates whether objects can be restored from
                                                              multiple
 * backups (
          backup planes) in the same restore. If not, the user will be
 * warned that changing backup planes will cause objects marked for
                                                               restore
 * from a different backup time to be autimatically unmarked.
 */
#define RSTPI_OPTION_MASK_MULTI_PLANE               0x8
#define RSTPI_OPTION_MULTI_PLANE_SUPPORTED          0x8
```

```
/* The following flag indicates whether the plugin provides an
 * RSTPI_GetNecessaryMedia function to return the list of media needed
                                                                    to
 * restore the restorable objects currently marked.
                                           Otherwise the plugin must
 * maintain the ebvl_volidlist_ty * list in the restore_context
                                                             whenever
 * objects are marked and unmarked.
 */
#define RSTPI_OPTION_MASK_GET_MEDIA                 0x10
#define RSTPI_OPTION_SPECIAL_GET_MEDIA              0x10
#define RSTPI_OPTION_GENERIC_GET_MEDIA              0x00

/* The following flag indicates whether objects can be restored
                                                              through
 * a symmetrix ('SCSI')
 */
#define RSTPI_OPTION_MASK_SYMM_RESTORE              0x20
#define RSTPI_OPTION_SYMM_RESTORE_ALLOWED           0x20

/*********************************************************************
                                                                    *
 *        Plug-in API function definitions
*********************************************************************
                                                                   */

/********************************************************************
 * Identify:
 *
 *  This function is called once,
                                  to identify and validate the plug-in with
 *  regard to the operating restore engine.
                                              The version number is checked
 *  for compatibility with the restore engine,
                                             and the optional features
 *  of the plug-in are specified.
 *
 * Parameters:
 *  pi_defs (
            O) - address of the structure identifying the plugin to the
 *               restore engine. Its fields consist of:
 *      version  - Version of the plugin header that the library was
                                                              built with
 *      name     - Name of the backup application (
                                                 64 byte buffer address)
 *      options  - Bit mask identifying the optional plug-in features
                                                               supported
 *          The bit definitions for this parameter (
                                             RSTPI_OPTION_MASK...) are
 *          defined above.
 *      wi_types  - pointer to array of workitem types supported by the
                                                                  plugin
 *      num_types - number of witypes in the witypes array
 *
 * Returns:
 *       E_SUCCESS          on success
 *       EP_RB_RECOVER_xxx  on error
 *
 *******************************************************************/
eerrno_ty RSTPI_Identify( const struct pluginIDdata **pi_defs );

/*******************************************************************
 * Initialize:
```

```
 *
 *  This function is called once, to allow the plug-in to perform its
 *  'local' initialization.
                               The restore context will have already been
 *  initiailized with the config file parsed,
                                               and the human user fields
 *  set.
 *
 * Parameters:
 *  context (I) - Pointer to the restore context
 *
 * Returns:
 *        E_SUCCESS          on success
 *        EP_RB_RECOVER_xxx  on error
 *
 *******************************************************************/
eerrno_ty RSTPI_Initialize( restore_context *context );


/*******************************************************************
 * Finish:
 *
 *  This function is called to allow the plug-in to clean up its
                                                           internal
 *  storage when a restore session is ending.
 *
 * Parameters:
 *  context (I) - Pointer to the restore context
 *
 * Returns:
 *        E_SUCCESS          on success
 *        EP_RB_RECOVER_xxx  on error
 *
 *******************************************************************/
eerrno_ty RSTPI_Finish( restore_context *context );



/*******************************************************************
 * Get Top Level Objects
 *
 *  This function is called to retrieve the configurable backup
                                                        objects (work
 *  items for network backups and work item sets for Symmetrix Connect
                                                           backups,
 *  which are restorable for the given client.
 *
 *  It is a GOAL of this routine to return all objects ever backed
 *  up successfully.  For network backups, though,
                                               it only looks in the config
 *  file for 'top level objects' of the given client.
 *
 *  While the restore API will be called repeatedly to retrieve a
                                                           maximum
 *  number of items on each call,
                                  this plug-in call must retrive the whole
 *  set of applicable backup objects.
                                    The generic restore service library
 *  will manage the composite list of top level objects from all
                                                        backup apps.
 *
 * Parameters:
 *   context        (I)  - Pointer to the restore context
 *   sourceHost     (
          I)  - the name of the host whose backups are being restored
 *   topLevObjs     (O)  - ptr to linked list of Top Level Objects
```

```
 *   numberEntries (
                  O)  - the real number of objects returned in the list
 *
 * Returns:
 *        E_SUCCESS          on success
 *        EP_RB_RECOVER_xxx  on error
 *
 *******************************************************************/

eerrno_ty
RSTPI_GetTopLevelObjects( restore_context          *context,
                          const char               *sourceHost,
                          struct RSTRPC_tlo_list   **topLevObjs,
                          short                    *numberEntries );

/**************************************************************
 *  Set TopLevel Object:
 *
 *  This function is called to notify the plug-in that one of its top
                                                                level
 *  objects has been selected (for browsing and marking). The plug-in
 *  should perform whatever validation and initialization is needed to
                                                              prepare
 *  for browsing and marking this top level object.
 *  If this call returns success,
                                then RSTPI_GetNextLevelObjects will be
 *  called to retrieve the highest level restorable objects for this
 *  backup object.
 *
 * NOTE: This function is responsible for setting the template_name
                                                                  and
 * saveset_thread elements of the restore context.
 *
 * Returns:
 *        E_SUCCESS          on success
 *        EP_RB_RECOVER_xxx  on error
 *
 * Parameters:
 *   context         (I)  - Pointer to the restore context
 *   backupObject    (I)  - the selected top level object
 *
 *******************************************************************/
eerrno_ty
RSTPI_SetTopLevelObject( restore_context              *context,
                         struct RSTRPC_top_level_obj  *backupObject
                                                                  );

/**************************************************************
 *  Get Next Level Objects:
 *
 *  This function is intended to allow retrieval of the children
 *  of a given parent object.
                              The caller specifies the parent object and
 *  whether or not to include bad files.  Even though the objects are
 *  returned in a linked list, there could conceivably be thousands of
 *  child objects, so the caller must specify the maximum number
 *  of children to return.  The caller is returned a token (
                                                    cookie) to allow
 *  continuing on the next call to thid function.
 *
 * Parameters:
 *   context         (I)  - Pointer to the restore context
 *   parentObject    (I)  - the parent object
 *   objectLevel     (
                   I)  - specifies whether parentObject is a top level or
 *                        container object
 *   objects       (
                O)  - a pointer to receive the start of the objects list
 *   cookie        (IO) - a place holder for the list position
 *   maxEntries    (I)  - the maximum number of objects to return
 *   numberEntries (
                O)  - the real number of objects returned in the list
 *   allowBadFiles (I)  - flag whether or not to include bad files
 *
 * Returns:
 *       E_SUCCESS          on success
 *       EP_RB_RECOVER_xxx  on error
 *
 ***************************************************************/

eerrno_ty
RSTPI_GetNextLevelObjects( restore_context          *context,
                           restorableObjectPtr      parentObj,
                           enum RSTRPC_ObjectLevel objectLevel,
                           struct RSTRPC_uro_list  **objects,
                           long                              *cookie,
                           const long               maxEntries,
                           long
                                                    *numberEntries,
                         const boolean_ty         allowBadFiles );

/*********************************************************
 *  Clear Restore Context :
 *
 *  This function is called to notify the plug-in that its currently
                                                               selected
 *  top level object is no longer selected (
                                           for browsing and marking). The
 *  plug-in should perform whatever cleanup and memory deallocation is
 *  appropriate.
 *
 * Returns:
 *       E_SUCCESS          on success
 *       EP_RB_RECOVER_xxx  on error
 *
 * Parameters:
 *   context          (I)   - Pointer to the restore context
 *
 ***************************************************************/
eerrno_ty
RSTPI_ClearRestoreContext( restore_context *context );

/***************************************************************
 * Submit
 *
 * This function creates a submit object from the currently marked
 * restorable objects.
                          The ID of the created submit object is passed to
 * EDMRST_Start to begin execution of the restore.
 *
 * Parameters:
 *
 * context        (I) - Pointer to the restore context
 * hostName       (I) - host to restore to (only if inPlace == False)
 * policy         (I) - The overwrite policy to use
 * inPlace        (
              I) - Flag if the restoral is to be in original locations
 * directory      (I) - directory to restore to (
                                            only if inPlace == False)
 * transport       (
         I) - Indicator of transport the restoral is to be over (SCSI
 *                    or network)
```

```
 * submitObjIDptr(
                IO) - ID of the submit user object created to describe
 *                  the restore
 * progressCB    (
               I) - pointer to callback function to report progress and
 *                  test for cancellation
 *
 ***************************************************************/
eerrno_ty RSTPI_Submit( restore_context         *context,
                          const char              *hostName,
                          const OverwritePolicy   policy,
                          const boolean_ty        inPlace,
                          const char              *directory,
                         const RestoreTransport   transport,
                         int                      *submitObjIDptr,
                          RSTPI_SubmitProgressProc progressCB );

/***************************************************************
 * StartRestore
 *
 * This optional function begins the application-specific execution of
                                                                      the
 * restoral of the objects in a submit object.  The quitTest callback
 * function must be called periodically to check for cancellation of
                                                              the restore.
 *
 * Parameters:
 *
 * context          (I) - Pointer to the restore context
 * submitObjectID (
                I) - ID of the submit object which describes the restore
 * quitTestCB      (I) - function to call to check for the quit signal
 *
 ***************************************************************/
eerrno_ty RSTPI_StartRestore( restore_context     *context,
                              int                 submitObjectID,
                              RSTPI_QuitTest      quitTestCB ) ;


/***************************************************************
 * Find Routines:
 *
 *  RSTPI_FindRestorableObjects and RSTPI_GetFindResults
 *
 *  These two functions allow the user to find restorable objects.
                                                                 Returned
 *  is a linked list of found objects.
                                       Each entry in the list contains a
 *  pointer to a restorable object,
                                   and the backup time associated with the
 *  object.
 *
 *  The find operation is performed by the asynchronous thread of the
                                                                  Restore
 *  engine, with the RSTSL_FindRestorableObjects function calling
 *  RSTPI_FindRestorableObjects if a plugin has its own find logic.
                                                        A second RPC call,
 *  RSTPI_GetFindResults, is used to test for completion of the find,
                                                                   and to
 *  signal cancelation.  A callback function pointer is passed into
 *  RSTPI_FindRestorableObjects to allow testing for cancelation of
                                                                the find,
 *  and to pass back progress information (
                                   a count of the objects found so far).
 *  The second service library function, RSTPI_GetFindResults,
                                                         is also used to
 *  retrieve the output of the find operation.
 *
 * RSTPI_FindRestorableObjects Parameters:
 *
 *   context          (I)  - Pointer to the restore context
 *   searchCriteria (I)  - The criteria used for the search
 *   intr_callback  (
                  I)  - Function to check for cancel and return progress
 *
 *******************************************************************/
eerrno_ty
RSTPI_FindRestorableObjects( restore_context            *context,
                             EBREC_SearchCriteriaRec
                                                          *searchCriteria,
                             RSTPI_FindProgressProc      intr_callback
                                                                       );

/*******************************************************************
 *
 * RSTPI_GetFindResults Parameters:
 *
 *   context        (I)  - Pointer to the restore context
 *   cancel         (I)  - requests cancellation of the find (if TRUE)
 *   maxEntries     (I)  - the maximum number of objects to return
 *   foundObjs      (O)  - a pointer to a linked list of found objects
 *   numberEntries (
                  O)  - the real number of objects returned in the list
 *   cookie         (IO) - a place holder for excess found objects
 *
 *******************************************************************/
eerrno_ty RSTPI_GetFindResults( restore_context     *context,
                                const boolean_ty    cancel,
                                const long          maxEntries,
                               struct RSTRPC_found_obj_list
                                                     **foundObjects,
                               long                 *numberEntries,
                               long                 *cookie );


/*******************************************************************
 * Get Top Level Templates:
 *
 *  This function is required to retrieve the templates with which a
                                                                  backup
 *  object could have been backed up.
 *
 * Parameters:
 *   context         (I)  - Pointer to the restore context
 *   topLevelObj     (I)  - the top level object
 *   templates       (
                  O)  - pointer to the start of the list of templates
 *   numberEntries (
              O)  - the real number of templates returned in the list
 *
 *******************************************************************/

eerrno_ty
RSTPI_GetTopLevelTemplates( restore_context              *context,
                             struct RSTRPC_top_level_obj *topLevelObj,
                            struct RSTRPC_name_list      **templates,
                            short
                                                      *numberEntries );

/*******************************************************************
 * Does Alternate Exist
 *
```

```
 *  This routine determines if an alternate trailset exists for the
 *  given template.
 *
 * Parameters:
 *  context       (I) - Pointer to the restore context
 *  templateName (I) - The name of the template to look for
 *  exists        (
              O) - Return flag for whether or not the alternate exists
 *
 *********************************************************************/

eerrno_ty
RSTPI_DoesAlternateExist( restore_context          *context,
                           const template_name_ty templateName,
                          boolean_ty              *exists );

/*********************************************************************
 * Mark Object
 *
 *   The MarkObject operation takes a restorableObject and marks it,
                                                                    and
 *   possibly its descendant files for restoral based on the input
                                                               criteria.
 *   Since the RSTPI_MarkObject call is an asynchronously executed
                                                               operation
 *   in the Restore Engine that performs the marking,
                                                       this function must
 *   periodically check for user-signalled cancelation,
                                                     and update progress
 *   data using the progress callback function argument.
 *
 * NOTE: This functions is responsible for keeping the volumes needed
                                                                    list
 *       (ebvllist) element of the restore context up to date.
 *
 * Parameters:
 *
 *  context        (I) - Pointer to the restore context
 *  thisObject     (I) - The restoral object;
                                          can be a leaf object (e.g. a
 *                   file), or a container object (
                                                  e.g., a directory).
 *  allowBadfiles (I) - allows marking of files of state BADDATA.
 *  descend        (
             I) - Should mark operation descend to operate on the content
 *                   of container objects.
 *  BadFilesCount (O) - returns the file count with BADDATA.
 *  PermDenyFilesCount (
              O) -- returns the file count with permission denied.
 *  fileMarked     (
             O) - return the total files marked after this mark occurred.
 *  lenMarkedFiles(
                O) - return the length of files marked after this mark
 *  dirMarked      (
               O) - return the total directories marked after this mark
 *                   occurred.
 *  otherMarked    (
           O) - return the total "other" files marked after this mark.
 *  progressCB     (
               I) - pointer to callback function to report progress and
 *                   test for cancellation
 *
 *********************************************************************/

eerrno_ty
RSTPI_MarkObject( restore_context *context,
                         struct RSTRPC_user_restorable_object *thisObject,
                        boolean_ty    allowBadFiles,
                        boolean_ty    descend,
                        unsigned long *BadFilesCount,
                        unsigned long *PermDenyFilesCount,
                        unsigned long *fileMarked,
                         u_hyper       *lenMarkedFiles,
                        unsigned long *dirMarked,
                        unsigned long  *otherMarked,
                        RSTPI_MarkProgressProc progressCB );

/*******************************************************************
 * UnmarkObject
 *
 *   The UnmarkObject operation takes a restorableObject and unmarks
                                                              it, and
 *   possibly its descendant files for restoral based on the input
                                                            criteria.
 *   Since the RSTPI_UnmarkObject call is an asynchronously executed
                                                            operation
 *   in the Restore Engine that performs the unmarking,
                                                 this plug-in function
 *   must periodically check for user-signalled cancelation,
                                                           and update
 *   progress data using the progress callback function argument.
 *
 * NOTE: This functions is responsible for keeping the volumes needed
                                                                 list
 *       (ebvllist) element of the restore context up to date.
 *
 * UnmarkObject Parameters:
 *
 *   context        (I) - Pointer to the restore context
 *   thisObject     (I) - The restoral object;
                                         can be a leaf object (e.g. a
 *                          file), or a container object (
                                                 e.g., a directory).
 *   BadFilesOnly  (
                   I) - allows unmarking ONLY of files of state BADDATA.
 *   descend        (
                  I) - Should unmark operation descend to operate on the
 *                          content of container objects.
 *   BadFilesCount (O) - returns the file count with BADDATA.
 *   fileMarked     (
                  O) - return the total files marked after this unmark
 *   lenMarkedFiles(
              O) - return the length of files marked after this unmark
 *   dirMarked      (
            O) - return the total directories marked after this unmark
 *   otherMarked    (
        O) - return the total "other" files marked after this unmark
 *   progressCB     (
              I) - pointer to callback function to report progress and
 *                          test for cancellation
 *******************************************************************/

eerrno_ty RSTPI_UnmarkObject( restore_context      *context,
                              struct RSTRPC_user_restorable_object
                                                         *thisObject,
                             const boolean_ty     BadFilesOnly,
                             const boolean_ty     descend,
                             unsigned long        *BadFilesCount,
                             unsigned long        *fileMarked,
                              u_hyper              *lenMarkedFiles,
                             unsigned long        *dirMarked,
                             unsigned long        *otherMarked,
```

```
                                         RSTPI_MarkProgressProc progressCB );

/*******************************************************************
 * Is Object Marked
 *
 *  This function determines if a restorable object has been
 *  marked for restoration.
                             It is intended to allow the user to determine the
 *  current restore markings for the restorable objects at the same
                                                            object tree
 *  hierarchy level,
                   i.e. objects that have the same parent restorableObject.
 *
 * Parameters:
 *
 * context       (I) - Pointer to the restore context
 * thisObject   (
            I) - The restoral object to be checked: can be a leaf object
 *                     (e.g. a file), or a container object (
                                                    e.g., a directory).
 * marked       (O) - boolean to receive the marked(1) / unmarked(
                                                             0) result
 *******************************************************************/

eerrno_ty
RSTPI_IsObjectMarked( restore_context                        *context,
                      struct RSTRPC_user_restorable_object
                                                           *thisObject,
                     boolean_ty                              *marked );

/*******************************************************************
 * Is Object Markable
 *
 * Function Description:
 *     Returns TRUE if the specified object is markable by the user,
 *     returns FALSE if it is not. This function applies only to
 *     container (directory) and leaf (file) objects.
 *
 * Parameters:
 *       context           - (I) Pointer to the restore context
 *     thisObject - (I) ptr to the restorableObject in question
 *
 * Return:
 *     TRUE - the specified object is markable by the user;
 *     FALSE - the specified object is not markable by the user;
 *
 *******************************************************************/
boolean_ty
RSTPI_IsObjectMarkable(
                       restore_context                        *context,
                         struct RSTRPC_user_restorable_object
                                                        *thisObject );

/*******************************************************************
 * Get Necessary Media:
 *
 *  This OPTIONAL function is provided to allow retrieval of the
 *  media necessary to restore the currently marked objects.
                                                        If a plugin does
 *  not supply this function,
                         then it must maintain the ebvllist volume list
 *  attached to the restore context whenever mark and unmark are
                                                          called, so that
 *  the generic RSTSL_GetNecessaryMedia function can retrieve the
                                                             media list.
 *
 * Parameters:
 *  context        (I) - Pointer to the restore context
 *  objects        (
            O) - address to return a pointer to the list of objects in
 *  numberEntries (
           O) - the real number of media objects returned in the array
 *
 *****************************************************************/
eerrno_ty RSTPI_GetNecessaryMedia( restore_context *context,
                                   struct RSTRPC_media_list **objects,
                                   short          *numberEntries );

/*****************************************************************
                                                                **
 *
 * Get All Backup Times
 *
 * Function Description:
 *      Retrieve the dates of the backups within the time range
 *      specified by the caller.
 *
 * Parameters:
 *      context              - (I) Pointer to the restore context
 *      startTime       - (I) include no earlier than this date
 *      endTime         - (I) include no later than this date
 *      flags           - (
                I) Backup constraint flags: e.g. full-only/partial-ok
 *      timesList       - (O) ptr to linked list of times
 *      numEntries      - (O) count of times returned
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *
 *****************************************************************/

eerrno_ty
RSTPI_GetAllBackupTimes( restore_context *context,
                          const time_t    startTime,
                         const time_t     endTime,
                          RSTRPC_backup_flags_ty flags,
                         struct RSTRPC_time_list **timesList,
                         short            *numEntries );

/*****************************************************************
                                                                **
 *
 * RSTPI_GetCurrentBackupTime
 *
 * Function Description:
 *      Retrieve the time of the backup that the current restore
                                                          context
 *      is set to and return it in the preallocated buffer.
 *
 * Parameters:
 *       context          - (I) Pointer to the restore context
 *      bkupTime - (O) the time of the backup
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *      EP_RB_RECOVER_INVALOP    - call issued out of sequence
 *      EP_RB_RECOVER_BAD_ARGS  - invalid input argument
 *      EP_RB_RECOVER_NO_CURR_BACKUP     - no valid backup currently
 *
 *****************************************************************/

eerrno_ty
```

```
RSTPI_GetCurrentBackupTime( restore_context *context,
                              time_t          *bkupTime );

/*****************************************************************
                                                                **
 *
 * Set Backup For Time
 *
 * Function Description:
 *      Switch to the backup plane of the specified time,
                                                  or the most recent
 *       that is before the specified time,
                                    if an exact mactch is not possible.
 *
 * Parameters:
 *  context (I) - Pointer to the restore context
 *  forTime (I) - The time for which the backup is requested
 *  flags   (I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS                - operation completed successfully
 *      EP_RB_RECOVER_xxx       - backup plane cannot be found
 *
 *****************************************************************/

eerrno_ty
RSTPI_SetBackupForTime( restore_context          *context,
                        const time_t             forTime,
                        RSTRPC_backup_flags_ty  flags );

/*****************************************************************
                                                                **
 *
 * Set Previous Backup
 *
 * Function Description:
 *      Set the restore context to that of the previous backup with
                                                            respect
 *      to the current one.
 *
 * Parameters:
 *  context (I) - Pointer to the restore context
 *  flags   (I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS     - operation completed successfully
 *      EP_RB_NO_PREV_CATALOG - when at the first catalog
 *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                               file
 *                                      of the new catalog
 *
 *****************************************************************/

eerrno_ty
RSTPI_SetPrevBackup( restore_context             *context,
                     RSTRPC_backup_flags_ty     flags );

/*****************************************************************
                                                                **
 *
 * Set Next Backup
 *
 * Function Description:
 *  This routine must set the restore environment to the the next
                                                             backup
 *  of the current top level object.
 *
 *  Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
 *            I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *      EP_RB_RECOVER_NO_NEXT_CATALOG - when at the most recent
                                                          catalog
 *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                             file
 *                                    of the new catalog
 *      EP_RB_RECOVER_NO_CATALOG - when mcat_set_mcplane failed
 *
 *******************************************************************/

eerrno_ty
RSTPI_SetNextBackup( restore_context            *context,
                     RSTRPC_backup_flags_ty     flags );

/********************************************************************
                                                                  **
 *
 * Set First Backup
 *
 * Function Description:
 *      Set the restore context to that of the first backup plane.
 *
 * Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
 *            I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *
 *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
                                                          file of
 *                                    the new catalog
 *
 *******************************************************************/

eerrno_ty
RSTPI_SetFirstBackup( restore_context            *context,
                      RSTRPC_backup_flags_ty     flags );

/********************************************************************
                                                                  **
 *
 * Set Most Recent Backup
 *
 * Function Description:
 *      Set the restore context to that of the most recent backup
                                                          catalog
 *      plane.
 *
 * Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
 *            I) - Backup constraint flags: e.g., full-only/partial-ok
 *
 * Return Codes:
 *      E_SUCCESS       - operation completed successfully
 *
 *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the
```

```
                                                          file of
 *                                       the new catalog.
 *
 ********************************************************************/

eerrno_ty
RSTPI_SetMostRecentBackup( restore_context            *context,
                           RSTRPC_backup_flags_ty     flags );

/*********************************************************************
                                                                    *
 * Is There Prev Backup
 *
 * Function Description:
 *      Determine if a backup exists prior to the backup that is
 *      currently selected.
 *
 * Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
 *            I) - Backup constraint flags: e.g., full-only/partial-ok
 *      isThere  (O) - TRUE/FALSE that requested backup does exist
 *
 * Return Codes:
 *      E_SUCCESS                - operation completed successfully
 *      EP_RB_RECOVER_xxx        - when errors occur accessing
                                                          catalogs
 *
 *********************************************************************
                                                                   */

eerrno_ty
RSTPI_IsTherePrevBackup( restore_context          *context,
                         RSTRPC_backup_flags_ty       flags,
                         boolean_ty               *isThere );

/*********************************************************************
                                                                    *
 * Is There Next Backup
 *
 * Function Description:
 *      Determine if a backup exists after the backup that is
 *      currently selected.
 *
 * Parameters:
 *       context (I) - Pointer to the restore context
 *       flags   (
 *            I) - Backup constraint flags: e.g., full-only/partial-ok
 *      isThere  (O) - TRUE/FALSE that requested backup does exist
 *
 * Return Codes:
 *      E_SUCCESS                - operation completed successfully
 *      EP_RB_RECOVER_xxx        - when errors occur accessing
                                                          catalogs
 *
 *********************************************************************
                                                                   */

eerrno_ty
RSTPI_IsThereNextBackup( restore_context          *context,
                         RSTRPC_backup_flags_ty  flags,
                         boolean_ty              *isThere );

/*********************************************************************
                                                                    *
 * Is There Prev Backup For Time
 *
 * Function Description:
 *      Determine if a backup exists prior to the specified time
 *
 * Parameters:
 *      context (I) - Pointer to the restore context
 *      thisTime(I) - Time for the query
 *      flags   (
          I) - Backup constraint flags: e.g., full-only/partial-ok
 *      isThere (O) - TRUE/FALSE that requested backup does exist
 *
 * Return Codes:
 *      E_SUCCESS              - operation completed successfully
 *      EP_RB_RECOVER_xxx      - when errors occur accessing catalogs
 *
 ***********************************************************************
                                                                      */

eerrno_ty
RSTPI_IsTherePrevBackupForTime( restore_context          *context,
                                 const time_t             thisTime,
                                 RSTRPC_backup_flags_ty   flags,
                                 boolean_ty               *isThere );

/***********************************************************************
                                                                      *
 * Is There NextBackup For Time
 *
 * Function Description:
 *      Determine if a backup exists after to the specified time
 *
 * Parameters:
 *      context (I) - Pointer to the restore context
 *      thisTime(I) - Time for the query
 *      flags   (
          I) - Backup constraint flags: e.g., full-only/partial-ok
 *      isThere (O) - TRUE/FALSE that requested backup does exist
 *
 * Return Codes:
 *      E_SUCCESS              - operation completed successfully
 *      EP_RB_RECOVER_xxx      - when errors occur accessing catalogs
 *
 ***********************************************************************
                                                                      */

eerrno_ty
RSTPI_IsThereNextBackupForTime( restore_context          *context,
                                const time_t             thisTime,
                                RSTRPC_backup_flags_ty   flags,
                                boolean_ty               *isThere );

#ifdef __cplusplus
}
#endif

#endif   /* #ifndef H_RSLPIAPI */
```